NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE ARCHITECTURE FOR A MULTI-LEVEL REAL-TIME SYSTEM

by

Omer Korkut

September 1998

Advisors: Mantak Shing
Valdis Berzins

Second Reader: Michael J. Holden

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1998	Master's Thesis

4. TITLE AND SUBTITLE
SOFTWARE ARCHITECTURE FOR A MULTI-LEVEL REAL-TIME SYSTEM

5. FUNDING NUMBERS

6. AUTHOR(S)
Korkut, Omer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

When a real-time system has a mixed set of time critical tasks, including tasks with hard deadlines and tasks with soft deadlines, managing a mixed set of tasks in a timely manner becomes harder and requires a multi-level architecture. This thesis concentrates on building such an architecture.

The proposed architecture is based on the current Computer Aided Prototype System (CAPS) architecture, which only deals with hard real-time and non-time-critical tasks. Priority-based scheduling techniques are used along with Ada tasking to schedule different levels of tasks. Periodic hard real-time polling tasks are used to insert sporadic soft real-time tasks into the system. A method is developed to assign deadlines to soft real-time tasks dynamically. Two tasking packages are added to the system for scheduling and execution of soft real-time tasks. The Earliest Deadline First (EDF) algorithm is used dynamically to schedule soft real-time tasks.

A pilot prototype is developed to test the proposed architecture via a run-time monitoring package. The results show that the proposed system guarantees that all hard real-time tasks meet their deadlines and an acceptably small percentage of soft real-time tasks miss their deadlines.

14. SUBJECT TERMS
Real-Time Systems, Real-Time Scheduling, Hard Real-Time Systems, Soft Real-Time Systems, Dynamic Scheduling, Preemptive Scheduling, Priority-Based Scheduling, Ada 95, Prototyping, Uni-processor Scheduling

15. NUMBER OF PAGES
130

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited

SOFTWARE ARCHITECTURE FOR A MULTI-LEVEL REAL-TIME SYSTEM

Omer Korkut
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1998

Author: _____

Omer Korkut

Approved by: _____

Mantak Shing, Advisor

_____

Valdis Berzins, Advisor

_____

Michael J. Holden, Second Reader

_____

for Dan C. Boger, Chairman
Department of Computer Science

ABSTRACT

When a real-time system has a mixed set of time critical tasks, including tasks with hard deadlines and tasks with soft deadlines, managing a mixed set of tasks in a timely manner becomes harder and requires a multi-level architecture. This thesis concentrates on building such an architecture.

The proposed architecture is based on the current Computer Aided Prototype System (CAPS) architecture, which only deals with hard real-time and non-time-critical tasks. Priority-based scheduling techniques are used along with Ada tasking to schedule different levels of tasks. Periodic hard real-time polling tasks are used to insert sporadic soft real-time tasks into the system. A method is developed to assign deadlines to soft real-time tasks dynamically. Two tasking packages are added to the system for scheduling and execution of soft real-time tasks. The Earliest Deadline First (EDF) algorithm is used dynamically to schedule soft real-time tasks.

A pilot prototype is developed to test the proposed architecture via a run-time monitoring package. The results show that the proposed system guarantees that all hard real-time tasks meet their deadlines and an acceptably small percentage of soft real-time tasks miss their deadlines.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

To my thesis advisor, Dr. Shing, I would like to express my deepest gratitude for all your support, guidance, and confidence. You always made yourself available whenever I have needed you througout this challenging journey.

To Commander Holden, who took time out of his tight schedule to provide assistance to me, I would like to thank you for your patience and help.

To Dr. Berzins and Dr. Luqi, who helped me in various ways during the course of this research, I would like also thank you both for all your encouragment and guidance.

To my closest friends, Lieutenant Steve Norris and Major John Lawson, who helped me and my family feel ourselves at home even though we have been thousands of miles away from home for more than two years, I will never forget you, your friendship, and the help that you have provided.

To my daughter, Cemre, who waited too long for a good-night-kiss many nights while I was studying at the school, I hope that you will be in my shoes and understand some day.

To my beloved wife, Arzu, who has always stood by me, I owe you eveything. This would not be a success without you and without all your patience, support, and encouragement. I just cannot put my appriciation into words.

Finally, thank you God for helping me climb one more step at my life-stairs.

I. INTRODUCTION TO REAL-TIME SYSTEMS

A. REAL-TIME SYSTEMS

Real-time systems emerge from the idea of involving computers in various control applications of today's modern life, as computers become faster, cheaper, and more dependable. As a result of rapid evolution of real-time systems led by an immense research effort in the area, real-time computing is now used in quite a few important applications. The list of these applications includes space programs, avionics, air traffic control, medical applications (e.g., intensive care monitoring), nuclear power plant control, defense applications, intelligent vehicle highway systems, and process control.

Although there are a number of different definitions given for real-time systems, the most widely accepted one defines a real-time system as a system in which the correctness of the system depends not only the logical results, but also on the time at which the results are produced [STAN 96].

A real-time system is very complex compared to a conventional computing system, because it has a sophisticated interaction with the physical world. The most important and significant characteristics of real-time systems are timeliness, predictability, trustworthiness, and simultaneous processing [TIMM 98].

As the definition of real-time systems implies, timeliness is one of the most distinctive properties of real-time systems. A real-time system should meet the deadlines of its tasks; that is, the application is required to finish certain tasks within the time boundaries that it has to respect.

A real-time system must respond to unpredictable stimuli in a predictable manner so as to react timely to all possible events. Therefore, predictability arises as another important issue in real-time computing.

Real-time systems must be reliable since they are often used in safety critical systems. It is necessary that a real-time system environment can depend on the system itself. So, trustworthiness for real-time systems is as important as timeliness and predictability are.

In a real world application, it is most likely that several elements of the application happen simultaneously or interact with each other. A real-time system must be so designed that it can serve parallel activities concurrently and meet all deadlines. This is sometimes difficult for a uni-processor system. In this case, although real concurrency cannot be provided, a feeling of concurrency can be achieved using preemptive scheduling methods.

B. CLASSIFICATION OF REAL-TIME SYSTEMS

In respect of the support given by the system to execute applications within their time constraints, authorities in the real-time computing field distinguish real-time systems as belonging to one of two major groups: hard real-time systems and soft real-time systems.

1. Hard Real-Time Systems

Hard real-time systems are well defined. They are the ones where it is definitely imperative that responses occur within the specified deadlines [ABAW 97]. No tardiness is accepted under any circumstances in hard real-time systems. Late results are useless

and cause system failures. A failure in a hard real-time system creates catastrophic consequences. It is impossible to compensate the cost of a missed deadline.

Hard real-time systems are needed in some application areas, which require strict safety and timing constraints. These areas include air traffic control, flight control system of a combat aircraft, nuclear power plant control, and space program applications.

According to characteristics of hard real time systems, a deadline is considered as hard if and only if it must be met by the system under all circumstances. The time-utility function[1] of a hard deadline is illustrated in Figure 1.1.

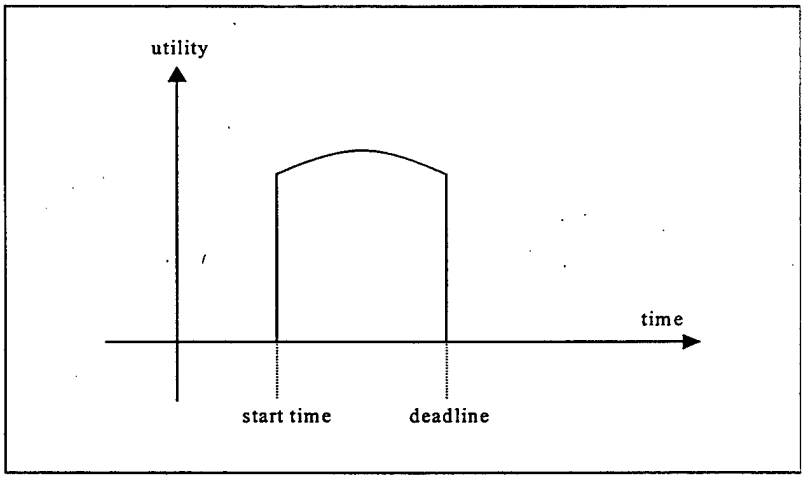


Figure 1.1. Time-utility function of a hard deadline

2. Soft Real-Time Systems

There is no conventional definition for soft real-time systems other than default as "not hard real-time". Missing some deadlines, by some amount, under some circumstances may be acceptable and may not cause a failure in a soft real-time system. Communications devices such as digital telephone exchanges are typical examples of soft real-time systems since they allow a certain amount of delay.

[1] Time utility functions illustrated in Figures 1.1-1.3 are also known as time-value functions [BURN 91].

Soft deadlines, similarly, are those that can be missed without jeopardizing the completeness of the system. However, missing the soft deadline of a task does incur a cost to the system and can be measured by the utility this task provides to the system. This means the later the response is, the less utility provides. The utility function of a soft deadline is shown in Figure 1.2. [BURN 91].

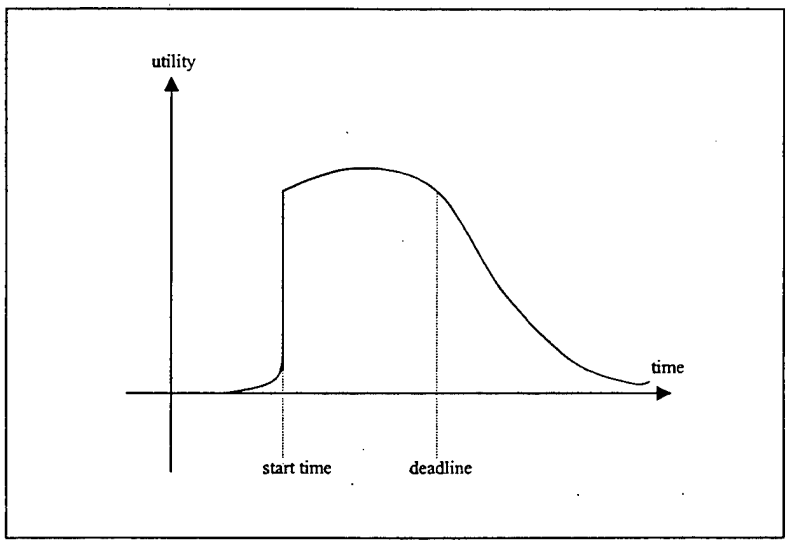


Figure 1.2. Time-utility function of a soft deadline

Furthermore, some tasks in soft real-time systems can have multi-valued timing constraints which specify the utility of tasks to the system in terms of best, better, worse, and worst completion times. With respect to this specification, the best completion time represents the most optimistic deadline which can be met by the system to gain the maximum utilization from a task, while the worst completion time defines the point at which a missed deadline starts to damage the system. Figure 1.3 illustrates the time-utility function of a multi-valued deadline [BURN 91].

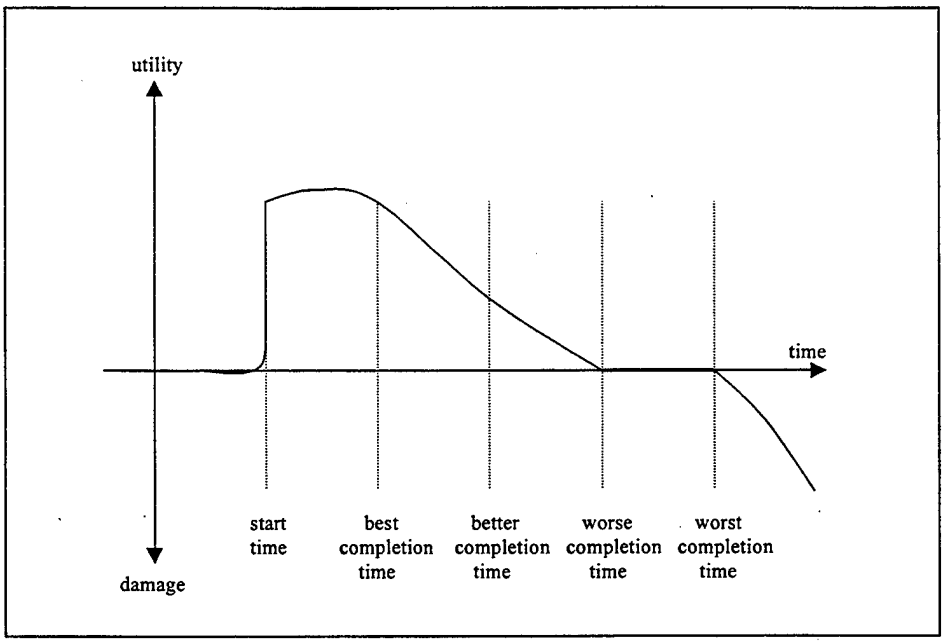


Figure 1.3. Time-utility function of a multi-valued deadline

3. Distinctions Between Hard and Soft Real-Time Systems

Although both hard and soft real-time systems have the major characteristics of real-time systems in common, each of them also possesses properties unique to those systems. The following list outlines some of the most important distinctions between hard and soft real-time systems.

- Hard real-time systems are well defined, but also are extreme and special cases. Soft real-time systems are much more complex, but at the same time are more general and common cases.
- Hard real-time systems have only tasks with deadlines defined in a binary logic, in contrast with soft real-time systems that have tasks with flexible timing constraints.

- Hard real-time systems usually use static resource management to serve tasks while tasks in soft real-time systems generally require dynamic resource management.
- Tasks in hard real-time systems usually have shorter deadlines than the ones in soft real-time systems, and tasks with soft deadlines almost always accommodate dynamic uncertainties.
- Hard real-time systems are more appropriate for unit-level regularity control applications whereas soft real-time systems tend to be more suitable for higher-level, higher-order control applications.

Besides the above classification of real-time systems, a system can have tasks with hard deadlines and tasks with soft deadlines. Such a multi-leveled real-time system architecture is the exact interest area of this research.

C. TASKS IN REAL-TIME SYSTEMS

Real-time systems deal with tasks with no timing constraints as well as they deal with tasks having hard and/or soft timing constraints. In order to service all these tasks, it is necessary to categorize tasks according to their common timing properties. First of all, tasks can be classified into two major groups: *time-critical* and *non-time-critical* tasks. Time-critical tasks are those having at least one timing constraint. Non-time-critical tasks are ones with no timing constraints at all.

While no further detailed classification for non-time-critical tasks is needed, time-critical tasks can be grouped as hard real-time tasks and soft real-time tasks depending upon how acute their deadlines are.

In addition to the categorization of tasks according to their timing constraints, grouping tasks in terms of their activation methods is also useful for proper scheduling of tasks. Tasks, in general, are classified as periodic, aperiodic, and sporadic tasks concerning their invocation methods [LUSH 96]. A complete classification is shown in Figure 1.4.

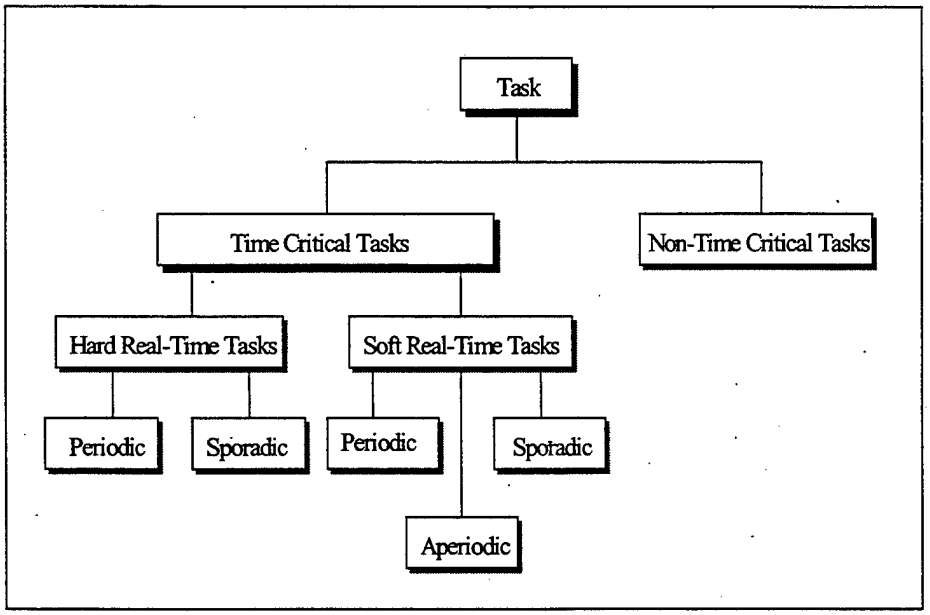


Figure 1.4. Classification of Tasks

Periodic tasks arrive on a regular basis and are characterized by their periods. They also have deadlines, which are often equal to their periods. All periodic tasks require a certain execution time per period. The execution time may be an average time or may be assigned as a result of a worst-case analysis [BURN 91].

As opposed to periodic tasks, aperiodic tasks arrive randomly. They are triggered by some external event. Aperiodic tasks may also have deadlines. However, random (sometimes like a burst) activation of aperiodic tasks allows any concentration of

activities, which makes it impossible to do a worst-case analysis and define a minimum period between any two tasks. Therefore, aperiodic tasks cannot have hard deadlines.

Sporadic tasks are in fact aperiodic tasks with specified minimum periods between any two arrivals of tasks. So, in addition to having all characteristics of aperiodic tasks, sporadic tasks can have also hard or soft deadlines.

After the introduction to real-time systems and tasks in real-time systems given in the first chapter, fundamental scheduling issues and current scheduling approaches involving real-time tasks are addressed in Chapter II of this thesis.

II. FUNDAMENTAL CONCEPTS IN REAL-TIME SCHEDULING

A. INTRODUCTION

Scheduling, in general, is the job of arranging activities that compete for access to a serially shared resource in a particular order. From the real-time point of view, time, in a sense, is also a resource which tasks contend for. With respect to this consideration, Ramamritham and Stankovic state that scheduling in real-time systems involves the allocation of resources and time to tasks in such a way that certain performance requirements are met [RAST 94].

Since there are a number of dimensions to the scheduling problem, there is no taxonomy for real-time scheduling on which the real-time community has reached an agreement. However, for the sake of the discussion addressed in this research, a modified version of the taxonomy presented by Cordeiro [CORD 95] is illustrated in Figure 2.1.

As shown in Figure 2.1, the scheduling theory in real-time systems can be divided into two major areas: hard real-time and soft real-time scheduling. This classification completely depends upon the characteristics of real-time systems, in other words, the tasks involved. Even though many basic algorithms and theoretical results have been developed for scheduling real-time systems, most of those results and algorithms involve tasks with hard deadlines, thus they are applicable in hard real-time systems. Because soft real-time systems are much more complicated than hard real-time ones, the soft real-time scheduling problem is more difficult than the hard real-time scheduling problem. Furthermore, comparatively little effort has been expended on soft real-time scheduling theory by the real-time computing research community. As a result, no easy constructive

and widely accepted algorithms appear to exist for any soft real-time cases, still leaving this topic an open research area.

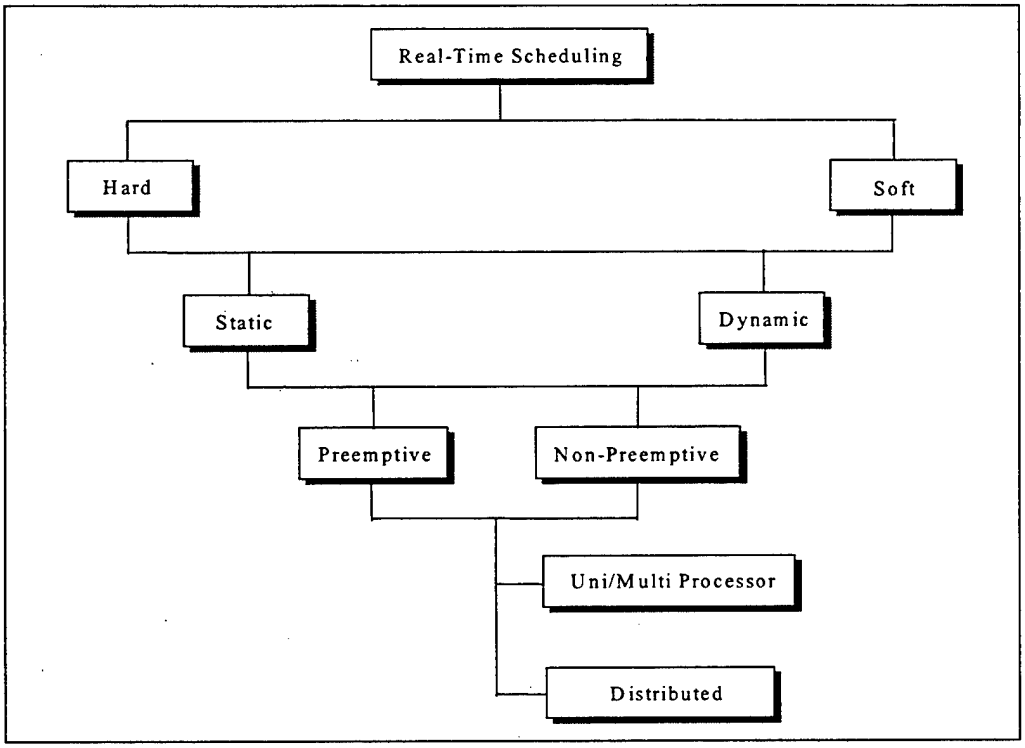


Figure 2.1. Real-Time Scheduling Taxonomy

B. STATIC VERSUS DYNAMIC SCHEDULING

Concerning the prior knowledge about tasks to be scheduled, scheduling algorithms can be categorized into two groups: static algorithms and dynamic algorithms [BURN 91].

The static approach requires that scheduling algorithm have complete prior knowledge of the task set including deadlines, execution times, precedence constraints, and future release times of each task in the set [SSDB 95]. Since the algorithm schedules tasks in advance, static scheduling requires little runtime overhead. Most of the time, static scheduling is achieved by either assigning fixed priorities to tasks, or setting up a

cyclic sequence of task execution. Fixed priorities associated with tasks do not change at run time, except the case of priority inheritance [JENS 98]. Even though fixed priorities are generally assigned to tasks in an ad hoc manner, there are some analytically proven techniques such as rate monotonic and deadline monotonic algorithms. These algorithms attach priorities to periodic tasks having hard deadlines. Currently, a large majority of real-time system applications are stable enough that they can be handled by static scheduling.

Even though the phrases "static scheduling" and "off-line scheduling" are usually used interchangeably by many of the researchers, they are totally different concepts. Off-line scheduling is an analysis that should always be done in the design phase of any real-time system regardless of whether the final algorithm is static or dynamic. According to the characteristics of real-time systems, or the tasks in real-time systems, dynamic or static scheduling algorithms can be created and used on-line or off-line. The important difference is the performance of the algorithm in each of these cases [SSDB 95].

Dynamic scheduling, in contrast with the static approach, produces schedules during execution. A dynamic scheduling algorithm has complete knowledge only of the currently active task set. The information about possible future arrivals is unknown to the algorithm at the time it is scheduling the current active set [SSDB 95]. Dynamic scheduling requires a skillful management of priority assignment to tasks. Application programs, rather than operating systems, usually perform dynamic priority assignments. Earliest deadline first (EDF) and least laxity first (LLF) algorithms are two well-known examples of numerous analytically based dynamic scheduling algorithms. These

algorithms can assign priorities to periodic, aperiodic, and sporadic tasks. Dynamic real-time scheduling is commonly used in military command and control applications.

C. PREEMPTION VERSUS NON-PREEMPTION

Preemption is another distinctive characterization in real-time system scheduling. Figure 2.1 illustrates that real-time scheduling can be either preemptive or non-preemptive. Preemptive scheduling algorithms can stop a task execution and resume its execution later without missing the task's deadline, but increasing the total elapsed time of the task. Preemption, if it is allowed, is the natural result of assigning priorities to tasks. A task with a higher priority can suspend another task with a lower priority and run until completion or until preempted by another third task that has a higher priority.

Non-preemptive algorithms, on the other hand, do not suspend tasks. Tasks in their running states finish their execution without preemption. Non-preemptive scheduling may be useful to achieve concurrency control for tasks executing inside a resource whose access is controlled by mutual exclusion [BURN 91].

A task with a lower priority can block a task with a higher priority while it is being executed in non-preemptive scheduling, which will not happen in preemptive scheduling. Therefore, the schedulability of a set of tasks by a non-preemptive algorithm is usually lower than that of its preemptive version. In other words, a set of periodic tasks can be scheduled by a preemptive scheduling algorithm if it also can be scheduled by the non-preemptive version of the same algorithm. However, the inverse may not be true and the probability that it is not true increases as the total utilization factor increases [TALI 94].

Although preemptive algorithms provide a better theoretical scheduling performance compared to non-preemptive ones, most of the time, this higher performance is obtained at the cost of higher scheduling overhead, such as frequently occurring context switches. Thus, in spite of their lower schedulability, non-preemptive algorithms, which have lower overhead, seem more advantageous from this perspective. Improvements in the speed of processors has also made it possible for non-preemptive algorithms to compete effectively "good enough" and "fast enough" for the overall system requirements.

Another problem with preemptive scheduling is *priority inversion*. Priority inversion is a situation in which a higher priority task is being blocked by a lower priority task. It is usually caused by the need to access shared resources, such as a critical section controlled by some synchronization mechanism. For instance, consider the example presented in Figure 2.2. A lower priority task L that previously has locked the critical section when a higher priority task H requests to get into the critical section. Task H is blocked by the task L inevitably. Furthermore, an executable medium priority task M wishing to execute after H is blocked will preempt L, and prolong the suspension of H indirectly. Figure 2.2 represent this priority inversion situation.

Priority Inheritance Protocol (PIP) and Priority Ceiling Protocol (PCP) are two well known protocols to prevent priority inversion.

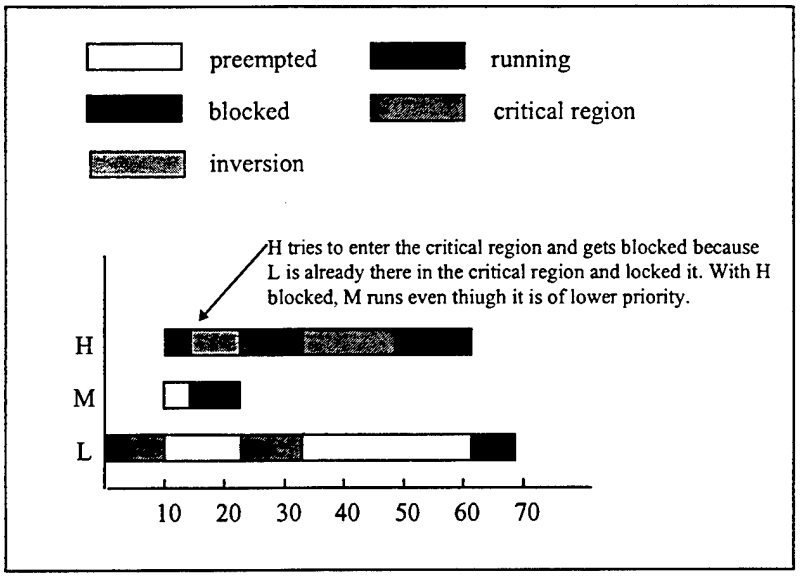


Figure 2.2. Priority Inversion Phenomenon

1. Priority Inheritance Protocol (PIP)

The PIP removes priority inversion dynamically. If a higher priority task requesting to enter into the critical region is suspended by a lower priority task already executing in the critical region, then the priority level of the lower priority task is promoted to the priority level of the higher priority task. This protects the higher priority task from being preempted by any other task whose priority is greater than the original priority of the preempting lower priority task. The effort of preventing priority inversion made by PIP is illustrated in Figure 2.3 using the same example in Figure 2.2.

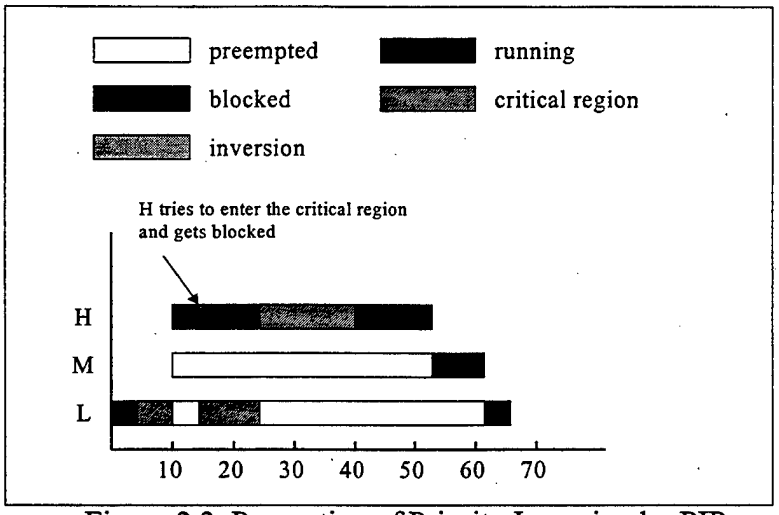


Figure 2.3. Prevention of Priority Inversion by PIP

There is a limit to the number of times that a task can be blocked by lower priority task in the PIP [SRLE 90]. If a task has n critical regions, then maximum number of times it can be blocked is n. Thus, in the worst case every critical region can be locked by a lower priority task. This given upper limit is too high and can likely lead to unacceptable results. Moreover, the possibility of chains of blocked tasks may cause deadlocks. A deadlock situation, which cannot be prevented by the PIP, is represented in Figure 2.4.

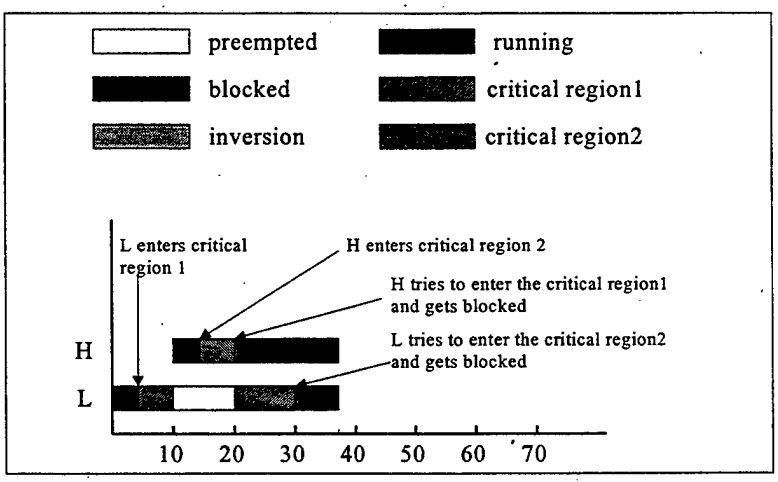


Figure 2.4. Inevitable Deadlock Situation in PIP

2. Priority Ceiling Protocol (PCP)

The PCP is proposed [SSDB 95] to solve the problems encountered in the PIP. In this protocol:

- All tasks have an assigned static priority.
- The ceiling of any critical region is defined as the priority of the highest priority task that currently locks or could lock that critical region.
- A critical region can be locked only if the priority of the requesting task is higher than the ceiling of all critical regions currently locked.
- In case of blocking, the task that holds the lock inherits the priority of the requesting task until it leaves the critical region.

One of the most visible benefits of the PCP is that a high priority task can be blocked at most once by any lower priority task before it enters into its first critical region. However, more tasks will experience this block. Another advantage of this protocol is prevention of deadlocks. Figure 2.5 shows how the PCP eliminates the deadlock situation in the example given in Figure 2.4.

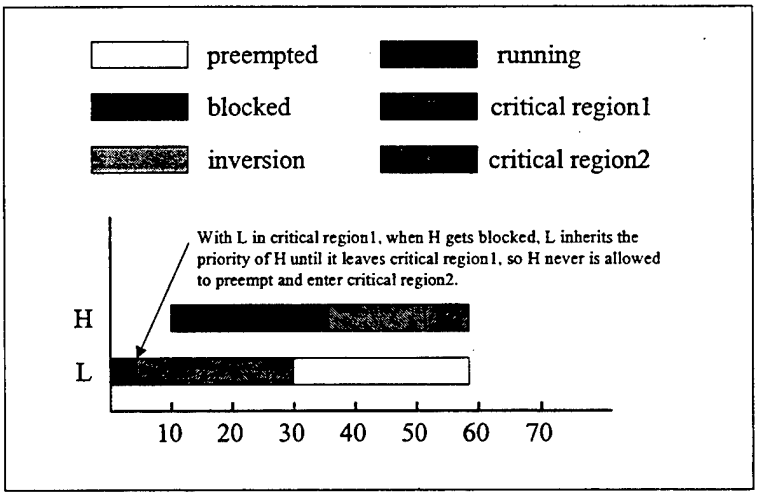


Figure 2.5. Prevention of Deadlocks in PCP

D. PERFORMANCE METRICS IN REAL-TIME SCHEDULING

Metrics used to measure the performance of scheduling algorithms in real-time systems are quite different from those used in non-real-time systems. For instance, while non-real-time systems typically use metrics such as minimizing the sum of the completion times of tasks in a set, minimizing the number of processors required by the application, or increasing the throughput, the main metric in real-time systems is achieving acceptable timeliness.

Metrics suggested for real-time systems vary according to the types of real-time systems, the requirements expected from these systems by real-world applications and the different types of tasks residing in the systems. Some of those metrics can be briefly explained as follows:

1. Meeting All Deadlines

This can only be achieved in a static environment, because prior knowledge of the task set is needed to be able to satisfy all deadlines. In this case, a schedule is found off-line to meet all deadlines and can be used on-line or off-line.

2. Maximizing Average Earliness

If there is more than one scheduling algorithm that meets all deadlines in a static scheduling environment, the goal should be choosing the best one among those algorithms. Maximizing average earliness can be used as a secondary metric in this case. The designer, then, prefers the algorithm that provides the best timely results [RAST 94].

3. Minimizing Average Tardiness

In case of non-existence of any scheduling algorithm that meets all deadlines, minimizing average tardiness of tasks that miss their deadlines can be used as a metric. Minimizing average tardiness is generally used in dynamic real-time environments, in which missing deadlines is an unavoidable consequence of lack of a priori knowledge about task sets to arrive.

4. Minimizing Missed Deadlines

When it is impossible to meet deadlines of all tasks, the best thing to do may be to meet as many deadlines as possible. Therefore, minimizing missed deadlines is often used as a metric in dynamic real-time systems.

Choosing the right performance measure for the right system is of utmost importance. Although the system requirements play an indicative role in determining the metrics to be used, in general, minimizing missed deadlines dominates the other metrics and is chosen as the primary metric for the systems that cannot meet all deadlines. An example of making a trade-off between minimizing average tardiness and minimizing missed deadlines is given in Figure 2.6. Even though minimizing missed deadlines is usually the most popular and dominant metric, there may be times when minimizing average tardiness is preferred depending upon the system requirements. The system designer should make the judgement whether lower average tardiness or fewer missed deadlines is the correct choice for that particular system.

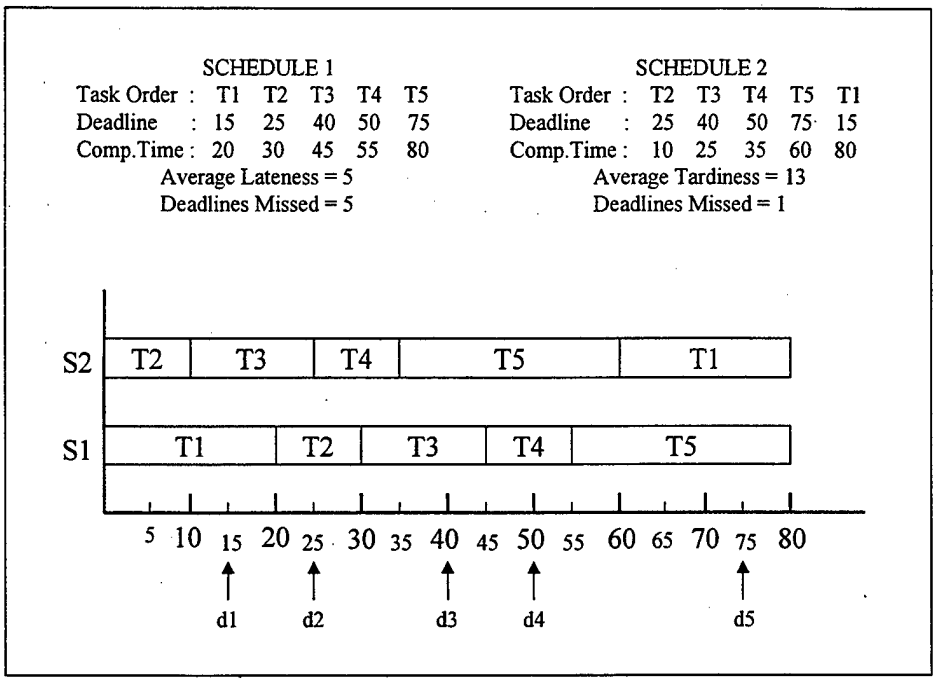


Figure 2.6. Minimizing Average Tardiness vs. Minimizing Missed Deadlines

E. CURRENT APPROACHES TO SCHEDULING ISSUES

In this section two basic task scheduling algorithms, *rate monotonic* and *earliest deadline first*, and their proven performances are briefly explained. Several other widely accepted algorithms are then addressed.

1. Rate Monotonic (RM) Algorithm

Scheduling of periodic tasks is one of the most important and attractive areas in real-time scheduling. The RM algorithm is commonly used to schedule sets of independent periodic tasks with preemption. It assigns static priorities to tasks according to their periods. Tasks with the shortest period get the highest priority and once the priorities are associated with tasks they remain unchanged. Liu and Layland proved that a set of n independent tasks can be scheduled by the RM algorithm if $\sum_{i=1}^{n} p_i/T_i \leq n(2^{1/n} - 1)$

where T_i and p_i are the period and worst case execution time respectively [LILA 73]. Assuming that all tasks have periods equal to their deadlines, they showed that the RM algorithm is an optimal fixed-priority scheduler for uni-processor scheduling environments, in the sense that any periodic task set schedulable by any other fixed priority scheduler is also schedulable by the RM scheduler.

When n in the above formula is large enough, the CPU utilization is bound to ln2. In other words, as long as the CPU utilization is less than 69% all tasks will meet their deadlines. The CPU utilization increases up to 88% when schedulability guaranteed for a random collection of tasks [LSDI 87].

Although every task must complete before the end of its period, i.e., deadline, there is no limitation as to when in the period it must be executed. This is because completion time of a lower priority task depends on whether it is preempted by a higher priority task arriving in the system. Therefore, unpredictable delays are inevitable in the RM policy, which is a big handicap [LIHE 96]. On the other hand, acceptably low overhead is the main advantage of the RM policy since the priorities are fixed and therefore implementation is easy.

2. Earliest Deadline First (EDF) Algorithm

As it is stated in the previous subsection the RM policy is a static priority assignment policy, and only applicable to periodic tasks, but, in the real world, we generally need dynamic priority assignment policies applicable to both periodic and aperiodic (sporadic) tasks. Here the EDF algorithm is often used.

The EDF algorithm depends on an intuitive priority assignment strategy. According to EDF, the highest priorities are associated with the tasks having the closest

deadlines. In contrast with static priorities, priorities assigned to tasks by the EDF are not fixed. For instance, a task's priority may be downgraded when another task with a tighter deadline arrives into the system. By doing that, the EDF guarantees that both tasks meet their deadlines. However, this brings run-time overheads into consideration and makes the EDF policy more expensive compared to any other static priority-driven algorithms.

For a set of independent periodic tasks, the following formula provides a necessary and sufficient condition for the schedulability of the tasks:

$$\sum_{i} C_i/P_i \leq 1$$

where C_i and P_i are the completion time and the period of task i respectively [LILA 73]. The EDF algorithm has also been shown to be optimal for periodic tasks under various stochastic conditions. It is optimal from the CPU utilization point of view when it is used to schedule aperiodic and sporadic tasks. However, investigations on scheduling sporadic tasks under the EDF algorithm shows that the EDF algorithm causes a quite large overhead in real-world applications, which sometimes is not acceptable [CHET 89].

3. Least Laxity First (LLF) Algorithm

Any function or characteristic of the tasks can be used to assign priorities to tasks. One of them is the laxity of a task, i.e., the certain amount of time a task can wait and still make its deadline. Similar to the EDF algorithm, in the LLF algorithm tasks with the least laxity get the highest priority and these dynamic priorities also may be change during the execution of the application [RAST 94].

4. Deadline Monotonic (DM) Algorithm

The DM algorithm is very similar to the RM algorithm. It assigns to each task a static priority inversely proportional to the task's deadline, i.e., tasks with the shortest deadline gets the highest priority. This static algorithm is also applicable to periodic task sets and is optimal for uni-processor environments when the tasks' deadlines are less than their periods [LEWH 82].

5. Slack Stealing Algorithm

The slack stealing algorithm was developed by Lehoczky and Ramos-Thuel as a solution to the problem of scheduling a mixture of hard periodic and soft aperiodic tasks [LERT 92]. Their approach proposes scheduling the task mixture in such a way that all periodic tasks make their deadlines and the response times for the aperiodic tasks are reduced as much as possible. Slack stealing is suitable for the systems scheduled using a static priority assignment policy such as the RM algorithm [BDTI 93].

Slack time can be defined as the amount of time that a periodic task leaves any resource idle in its period. In other words, it is the time that a task has before its deadline [BURN 91].

The slack stealing algorithm does not create a periodic server to schedule aperiodic tasks, unlike similar algorithms seeking for a solution to the same problem. Instead, it provides the time required by soft aperiodic tasks via a passive task, called slack stealer. The slack stealer created by the algorithm, when it is prompted, services aperiodic tasks by stealing as much slack time as it can from the periodic tasks without causing them to miss their deadlines.

Burns, Davis, and Tindell improved the static slack stealing algorithm developed by Lehoczky and Ramos-Thuel. The new algorithm is based on the old one, but it steals slack time from not only hard periodic tasks but also from hard sporadic tasks. Even though this dynamic approach is applicable to a wider class of scheduling problems and is proven to be optimal to improve response times, it is not feasible in practice because of the overhead it causes at run-time. The authors address this problem in their work in detail [BDTI 93].

III. THE COMPUTER AIDED PROTOTYPING SYSTEM (CAPS)

Prototyping is one of the most promising methods of improving programming productivity and the reliability of the software product [LUBY 88]. Prototyping also reduces the efforts made in software evolution activities [LUQI 89]. In the same context, rapid prototyping can be defined as the process of building and evaluating a series of prototypes.

CAPS is a software engineering tool that is specifically designed to develop software prototypes of real-time systems. CAPS supports the design, development, modification, and validation of prototypes as well as the analysis of system requirements. The Prototype System Description Language (PSDL) is a partially graphical prototyping language that is specifically designed for CAPS. PSDL equips CAPS with the abilities of setting timing and control constraints within a software system. A diagram of the CAPS development environment is shown in Figure 3.1.

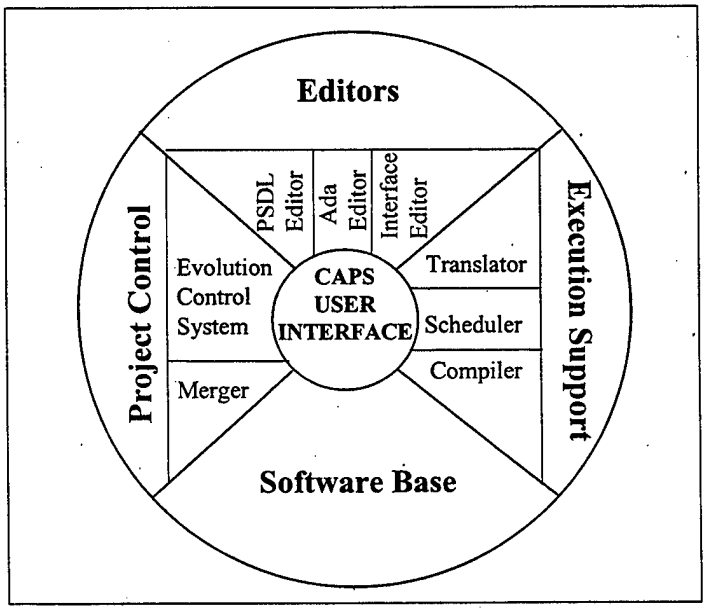


Figure 3.1. The CAPS Development Environment

CAPS is set of software engineering tools that are linked together by a common user interface. The CAPS development environment provides the necessary tools to enable the designer to rapidly develop, analyze, and modify real-time software systems. Assistance provided by CAPS to software engineers can be summarized as follows:

- Timing feasibility checking via the scheduler.
- Consistency checking and automated assistance for project planning, scheduling, designer task assignment, and project completion date estimation via the Evolution Control System.
- Design completion via the editors.
- Computer-aided software reuse via the Software Base

A. THE PROTOTYPE SYSTEM DEFINITION LANGUAGE (PSDL)

PSDL is designed to describe prototypes of real-time systems. It provides assistance in requirements analysis, feasibility studies, and the design of large real-time embedded systems. Prototypes built using PSDL are executable if a software base containing reusable software components in an underlying programming language such as Ada supports them [LUBY 88]. PSDL supports frequent design modifications by providing modularity, simplicity, reuse, adaptability, abstraction, and requirements tracing.

1. Computational Model

The PSDL computational model is based on a directed graph containing operators and data streams associated with timing and control constraints. Operators communicate with each other directly or indirectly via data streams. Each date stream carries values of

a fixed abstract data type as well as types already built into PSDL, such as the PSDL_EXCEPTION type. Operators are illustrated as vertices and data streams are illustrated as edges in the graph.

The PSDL computational model is formally represented as an augmented graph, G = (V, E, T(v), C(v)) [LUBY 88],

where:

- V is a set of vertices.
- E is a set of edges.
- T(v) is the set of timing constraints for each vertex v.
- C(v) is the set of control constraints for each vertex v.

a. Operators

A PSDL operator can be either a function or a state machine. The act of executing an operator is generally called firing, and it involves reading one data object from each input data stream, computing results only if the execution guard is satisfied, and writing at most one data object to each output data stream. If the output of an operator depends on only the current set of input values, then the operator represents a function. On the other hand, if the output depends on both the current set of input values and the internal memory (state variables) of the operator, then the operator represents a state machine.

PSDL operators can be categorized as atomic or composite operators. Operators that cannot be decomposed in terms of the PSDL computational model are

atomic operators. In contrast with atomic operators, composite operators can be expressed in terms of lower level operators with all their data and control networks.

PSDL operators are examined in two major parts: their specification and their implementation. The specification part consists of attributes defining the form of the interface, both formal and informal descriptions of the noticeable behaviors of the operator, and timing constraints. The implementation part plays a role in deciding whether the operator is atomic or composite.

In addition to conventional PSDL operators, there is a special type of operator called terminator. Terminators represent simulation of external systems and are not parts of the delivered software, and hence do not use CPU time in the designed system.

b. Data Streams

Data streams connect a set of one or more operators to a set of one or more operators. The operators which streams are originated from are called producers while operators which streams go into are called consumers. Data streams carry a sequence of data values from producer operators to consumer operators. Data streams are classified as data-flow streams and sampled streams.

The data trigger of a consumer operator determines the type of a data stream: if the consumer operator fires on every occurrence of data on a data stream, then the stream is a data-flow stream; otherwise it is a sampled stream. Typical representation of two operators connected via a stream in the PSDL graphic editor is illustrated in Figure 3.2.

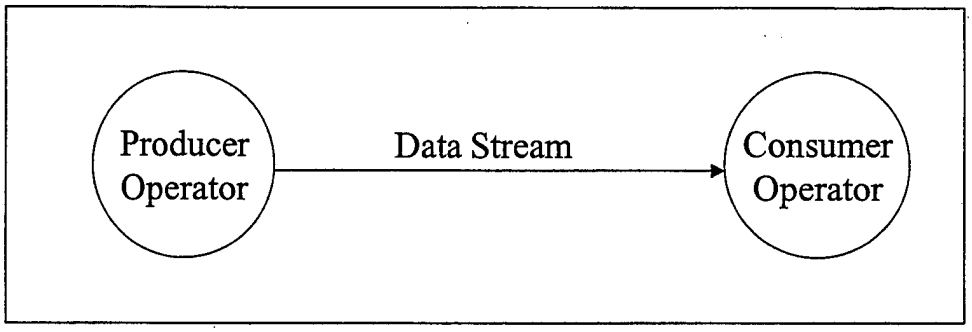


Figure 3.2 Two operators connected via a stream in PSDL graphical editor

A data-flow stream promises that no data is lost or duplicated [LUBY 88]. Data-flow streams can be considered as FIFO queues with a length of one. Any attempt by a producer to place a data value into a data flow stream before the data already existing in the queue is read and consumed by the consumer operator will cause overflow and results in failure of the attempt. Similarly, if a consumer tries to read from an empty data-flow stream, this attempt will fail because of underflow. It is appropriate to use data-flow streams if each data value represents a unique transaction or request that must be acted on only once. Data-flow streams could also be used in the cases that the data-yielding rate of the producer and the data-consuming rate of the consumer closely match.

In contrast to data-flow streams, sampled streams do not guarantee against data replication and data loss. Sampled streams also hold only one data at a given time just as data-flow streams do, but the data value read from a sampled stream by the consumer is not destroyed until the producer puts another data value into the stream. On the other hand, if the producer tries to place a new data value into a sampled stream the old value in the stream is thrown away without considering whether it has been used or not by the consumer. Therefore, sampled data streams are generally used in cases where only the most recent information is meaningful, such as simulating continuous streams of information [LUBY 88].

c. State Streams

If an operator is a state machine, then state streams are used to deal with state information of the operator, which are expressed as self-loops or cycles in the PSDL graph. Before scheduling, state streams are removed from the graph because a PSDL graph is schedulable only if it is a directed acyclic graph (DAG), a directed graph without any cycles. State streams are declared in the specification part of the parent operator to indicate that it is a state machine rather than a function and the declaration of a state stream must supply an initial value for the stream. State streams can be either data-flow streams or sampled streams depending upon the triggering condition of the consumer operator.

d. Types

PSDL types used in CAPS prototypes consist of user-defined ADTs and a subset of immutable built-in types of the Ada programming language. PSDL types, like PSDL operators, have a specification and an implementation part and can be implemented either in PSDL or Ada. Types in PSDL can be associated with operators. Types implemented in Ada are taken care of by an Ada package that defines a private type and a subprogram for each operator on that type.

e. Exceptions

PSDL exceptions are values that can be transmitted along data streams and their type in PSDL is represented by a built-in abstract data type (ADT) called "PSDL_EXCEPTION". This ADT has operations to create an exception with a given

name by the designer, detect whether a value is an exception with an assigned name, and determine whether a value belongs to any data type other than PSDL_EXCEPTION (keyword "NORMAL" is used to determine that). While the prototype is being executed, undeclared exceptions of the programming language exceptions (for the current version of the CAPS, it is Ada) are transformed into PSDL_EXCEPTION.

2. Control Constraints

There is no explicit control algorithm in PSDL to specify control aspects of an operator. Instead, PSDL uses control constraints to describe control aspects of an operator. Whether an operator is periodic or sporadic, triggering conditions, timers, execution guards, and output guards are control aspects to be specified.

a. Periodic and Sporadic Operators

Time-critical operators in PSDL are classified as periodic or sporadic operators. Periodic operators are triggered by the scheduler at approximately regular time intervals according to their periods. They start execution somewhere after the beginning of their periods, and complete execution before a specified deadline, which by default is the end of their periods.

If there is no specified period for a time-critical operator, that operator is a sporadic operator. Sporadic operators are triggered by the arrival of new data, which makes them less predictable compared to periodic operators.

b. Data Triggers

Every time-critical operator in PSDL must have a period or a data trigger, or both. If a periodic operator also has a data trigger, in that case the data trigger servers as an input guard and the operator is executed conditionally with respect to the trigger [LUBY 88]. There are two kinds of data triggers that a PSDL operator can possibly have and they are defined by "TRIGGERED BY ALL" and "TRIGGERED BY SOME" keyword expressions. These two data triggers are illustrated and explained in detail by the examples below.

OPERATOR *R* TRIGGERED BY ALL *a, b, c*

OPERATOR *S* TRIGGERED BY SOME *x, y*

In the first example, the operator *R* cannot fire unless there is new data on all three of the input data streams, *a*, *b*, and *c*, which are generally called the *triggering set*. This kind of trigger promises that the output of the operator depends on fresh input data and can be used for synchronization purposes. On the other hand, the second type of data trigger presented in the second example allows operators to fire when new data arrives in either the *x* or *y* input data streams. It is appropriate to use this type of data trigger to keep software estimates of sensor data up to date.

c. Timers

A PSDL timer is a special type of abstract state machine used as a software stopwatch to measure the time interval the system spends in a given state, or the length of time between specified events. Timers are declared in the implementation part of a composite operator. Timers provide a non-local means of control by their three

special control constraints: "START TIMER", "STOP TIMER", and "RESET TIMER". A timer is visible in the module in which it is declared. Furthermore, if a timer is declared in a composite module, then it is also visible in the components of the composite [LUBY 88].

d. Execution Guards

Execution guards are conditional statements that are used instead of, or in addition to, data triggers to authorize a PSDL operator to fire. An execution guard can depend only on data coming from any input stream and data from timers. The following examples illustrate execution guards with and without data triggers.

OPERATOR *A* TRIGGERED BY ALL x, y, z IF $x > y$

OPEARTOR *B* TRIGGERED IF $w < 10.0$

Even if an execution guard is not met, which causes the operator not to fire, the operator will consume the data values read from input streams.

e. Output Guards

Output guards are the determining factor in deciding whether the result of the computation done by the operator is going to be output or not. Unless the predicate in the guard evaluates to TRUE, no output is placed into the output stream after the operator fires. The filtering process achieved by the output guard, if the predicate evaluates to FALSE, does not affect the firing of the operator. A PSDL operator fires regardless of whether its output is written into the output stream or not. Output guard predicates can depend only on the data coming from input streams, the output value produced by the operator itself, and data from timers.

3. Timing Constraints

One of the essential tasks in designing a real-time system is specifying the timing constraints. Schedulability of a set of timing constraints is one of the most important problems to be solved in a real-time system. The types of timing constraints associated with PSDL operators and streams are represented in Table 3.1.

Constraints	Abbreviation	Applies to	Constrains	Default
Maximum Execution Time	MET	Time Critical Operators	CPU Time	–
Period	P	Periodic Operators	Activation, Next Activation	–
Finish Within	FW	Periodic Operators	Activation, Completion	P
Maximum Response Time	MRT	Sporadic Operators	Activation, Completion	Heuristic
MinimumCalling Period	MCP	Sporadic Operators	Activation, Next Activation	MRT - MET
Latency	L	Streams	Write, Next Read	0
Minimum Period	MP	Streams	Write, Next Write	0

Table 3.1. Types of PSDL Timing Constraints "From Ref.[LUSH 96]"

The maximum execution time (MET) is the time needed by CPU to execute an operator under worst-case conditions. MET is expressed relative to the host hardware for the CAPS system and must be scaled by the scheduler if the target hardware for the prototype has a different execution speed. This constraint must be specified to all time-critical PSDL operators so that the scheduler can allocate enough CPU time to satisfy their deadlines [LUSH 96].

Both period and finish within (FW) are the only constraints applied to periodic operators. Period is the time interval between two consecutive activations of an operator, while FW represents the upper bound on the time interval between each activation and completion of execution of an operator. FW, indeed, is equal to the deadline of an

operator. The time interval represented by FW is also known as *scheduling interval* (SI). Even though periodic operators are activated at regular intervals equal to their periods, a delay can occur between the activation time and the actual starting time of the execution. This delay, which cannot be greater than the difference between MET and FW, is called the *slack* of the operator and controlled by the scheduler [LUSH 96]. Figure 3.3 shows the timing constraints for periodic operators.

Scheduling delay for a periodic operator is the duration between the beginning of the period of the operator and the time at which the operator reads from the input stream.

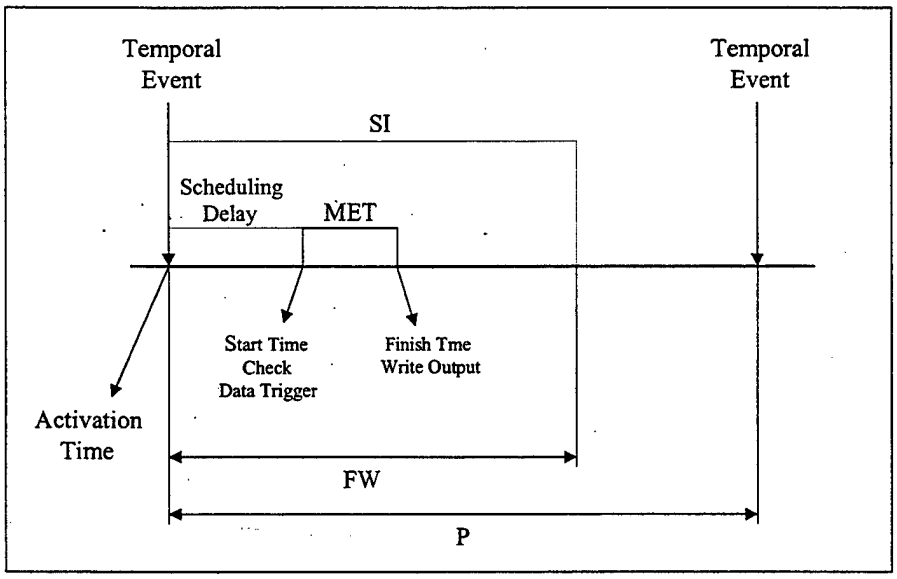


Figure 3.3 Timing Constraints for Periodic Operators

The maximum response time (MRT) associated with sporadic operators is the upper bound on the time interval between the arrival of new input data that meets all data trigger conditions for an operator and the completion of firing.

The minimum calling period (MCP) is the shortest allowable duration between two consecutive activations of a sporadic operator. The MCP does not constrain the behavior of the operator itself but it does constrain the behavior of the producers of the

triggering data values. Therefore, considering MCP as a "minimum satisfaction period" may be useful.

Before scheduling, every sporadic operator is transformed to an equivalent periodic operator, whose period is called as the triggering period (TP). Timing constraints for sporadic operators are illustrated in Figure 3.4.

Scheduling delay for sporadic operators, which is different from the scheduling delay for periodic operators, is the duration between the time the producer operator writes into the output data stream and the time the consumer reads the same data from the input stream.

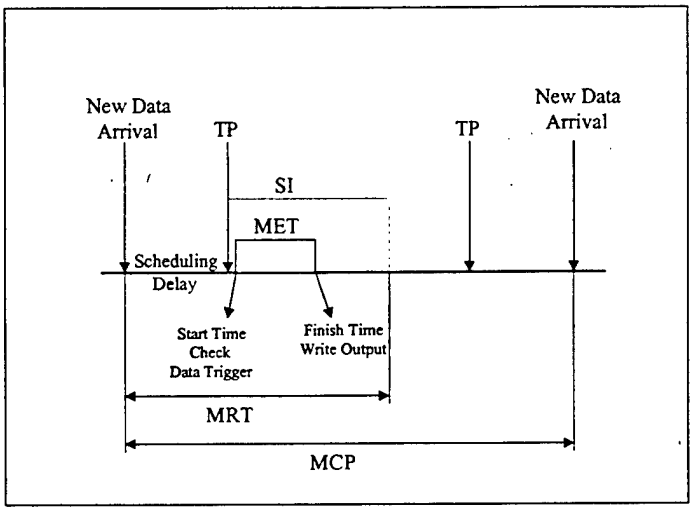


Figure 3.4 Timing Constraints for Sporadic Operators

The *latency* of a data stream is the longest allowable duration between the time a data value is written into a stream and the time the same data value becomes available to be read from the same stream. The scheduler can use the latency of a stream to estimate the network delay while simulating the worst case behavior.

Minimum period (MP) is a lower bound on the time interval between two successive write events on the same stream. Both MP and latency are external requirements and constrain the scheduler and the implementation.

B. CAPS TOOLS

1. Editors

a. PSDL Editor

The PSDL editor is one of the most important elements of the CAPS since it is used to build the prototypes. The PSDL editor enables the designer to create the CAPS data flow diagram and enter textual constraints via pull-down and pop-up text boxes. All control and timing constraints are assigned to operators and data streams using the PSDL editor.

b. The Text Editor

Even though the text editor is not an actual part of CAPS, a list of editors is integrated into CAPS as a text editing facility. Vi, Emacs, and the Verdix Ada Syntax Directed Editor are provided as choices given in the list. The user can choose any one of these editors to edit Ada code by using a pull-down menu.

c. The Interface Editor

The tool called Transportable Applications Environment Plus (TAE+) is available through CAPS to create and manipulate dynamic window-based graphical user-interfaces (GUIs) for prototypes. When the designer chooses the "single file" Ada code

generation option within TAE+ during creation of such user-interfaces via the TAE Workbench, the automatically generated TAE code is placed under the prototype directory in a file named <prototype_name>.RAW_TAE_INTERFACE.a.

d. The Requirements Editor

There is no advanced tool associated with the current version of the CAPS for editing or tracking requirements. A simple text editor is available in CAPS to edit requirement documents for prototypes. When the "Requirements" item is selected from the "Edit" pull-down menu a list of files with the suffix ".req" is represented. Then, the designer selects a file from the list and invokes the default text editor on that same file.

e. The Change Request Editor

A sophisticated change request editing or tracking tool has not been integrated into the current version of the CAPS. As the requirements editor, the "Change Request" item in the "Edit" pull-down menu shows the designer the list of the files with the suffix ".cr". The designer summons the default editor to edit a specified file by simply selecting that file from the presented list.

2. Execution Support

a. The Translator

The translator is the CAPS tool that achieves the conversion of a PSDL program to Ada packages, which serve as supervisor modules for the prototype. The CAPS translator augments the implementations of atomic operators and types with code

realizing the data streams and activation conditions, resulting in the underlying programming language, Ada, that can be compiled and run [LUBY 88]. However, the translator itself does not create Ada implementation packages either for atomic operators or user-defined types. It just generates code to link the components retrieved from the available software base or built by the designer. A successful translation is required to be able to schedule the prototype.

b. The Scheduler

In the current version of the CAPS, the scheduler demonstrates the schedulability of a prototype via generation of a static run-time schedule that enforces every hard real-time constraint with respect to the worst case analysis [LUSH 96]. The PSDL programs of the prototype provide the scheduler with information about timing constraints.

The CAPS scheduler creates two different schedules: one of them is a static schedule with a higher priority for time-critical operators (hard real-time operators), and the other is a dynamic schedule with a lower priority for non-time-critical operators. The static schedule tries to allocate CPU time for time-critical operators, and if it succeeds, every time-critical operator meets its deadline. The dynamic schedule, on the other hand, schedules non-time-critical operators into the slots which are not previously allocated, during run-time.

At the end of the scheduling attempt of a prototype, the scheduler gives a diagnostic report about the attempt regardless of whether the attempt was successful or not. If the scheduler cannot find a feasible schedule for the given timing constraints, the diagnostic information can be useful to the designer in the process of localizing and

eliminating the problems. A prototype must be successfully scheduled before it is compiled.

c. The Compiler

The CAPS uses the SunAda Ada compiler. The formal parameter lists in the modules that implement atomic operators must be in conformity with the CAPS interface conventions to achieve a successful compilation.

3. Project Control

a. The Evolution Control System (ECS)

Bigger systems can be prototyped faster by taking advantage of team-work. The ECS subsystem enables a group of designers to work on the same prototype development in a distributed environment. A design database (DDB) is used by the ECS to achieve the consistency of the prototype development data. At the same time, the ECS maintains a designer pool from which the prototype development tasks are drawn. The project manager leads the distributed prototype development in a step-by-step fashion using the ECS. The steps created by the manager are automatically scheduled and assigned to available designers.

b. The Merger

The CAPS merger provides the facility for automatic combination of two independently realized modifications to a base prototype. The merger warns the designer if a conflict occurs between two changes during the merge process. In case of a

successful merge of modifications, the merger yields a PSDL program for the newly created base prototype, which incorporates the changes made to each modified prototype.

4. The Software Base

A repository for reusable Ada and PSDL components and a mechanism to access these modules are provided and maintained by the CAPS software base. A designer can retrieve the module needed by browsing as well as querying. During the browsing process, the user can either use types or operators. For querying the components from the software base, available keywords and PSDL specifications can be used.

IV. ARCHITECTURAL ISSUES OF A MULTI-LEVEL REAL-TIME SYSTEM

A. MOTIVATION

A real-time system can have a mixed set of tasks, including tasks with hard deadlines, tasks with soft deadlines and tasks with no deadlines at all. Therefore, a multi-level real-time system must cope with such a mixed set of tasks. In this research, we consider a mixed set of tasks consisting of hard periodic tasks, soft sporadic tasks and non-time-critical tasks. The problem of scheduling such a mixed set of tasks is a hard and complicated issue.

First of all, the system must meet deadlines of all hard real-time tasks, because missing any hard deadline causes system failure. The real-time system should also meet as many soft deadlines as it can. Even though missing a soft deadline does not cause any serious damage to the system, missing all soft deadlines is not acceptable either. Therefore, such a multi-level real-time system should meet an acceptable number of soft deadlines defined by the system requirements to provide a satisfactory utilization level. At last, the real-time system should be able to execute non-time-critical tasks while it is scheduling and executing hard and soft real-time tasks in a timely manner. Although non-time-critical tasks have no deadlines, they may need to execute once in a while to satisfy the system requirements. Hence, starvation of non-time-critical tasks while hard and soft real-time tasks are executing may not be a desired system behavior.

The goal of this research is to develop a software architecture for the CAPS generated prototypes with multi-level real-time constraints like the one explained above

for a single processor platform. In this architecture, we will introduce some new techniques that are not included in the current CAPS generated control code. Suggested methods to develop the new architecture and the relevant capabilities of CAPS will be discussed in this chapter.

B. PRIORITY-BASED SCHEDULING

In order to have a better understanding of how current CAPS scheduler and the new architecture that will be suggested work, it is useful to review the concept of *priority-based scheduling* and Ada 95 programming language support for priority-based scheduling.

A priority value assigned to a task indicates the degree of urgency of that particular task and also is the basis for resolving competing demands of tasks for resources. Whenever tasks compete for processors or any other implementation-defined resources, the resources are allocated to the task with the highest priority unless otherwise specified [ALRM 95].

Assigning priorities to tasks to determine which task is selected for execution when more than one task is ready to execute is a well-known and accepted technique. Such technique is commonly known as priority-based scheduling. Even though priority-based scheduling has traditionally been more an issue for operating systems than for programming languages, few programming languages explicitly define priorities as part of their concurrency facilities. Ada 95 is one of a few languages that provide a relatively detailed and complete support for priority-based scheduling [ABAW 97].

1. Priority-Based Scheduling with Ada 95

Besides being defined as a core language, Ada 95 also has a number of annexes for specialized application domains. Although these annexes do not introduce any new language features, they define pragmas and library packages that must be supported if that particular annex is to be adhered to. The Real-Time Systems Annex (Annex D) of Ada 95 provides pragmas to allow to assign priorities to tasks [ABAW 97].

The following priority-related declarations reside in a library package called *System* in Ada 95:

SUBTYPE Any_Priority **IS** Integer **RANGE** *implementation-defined*;

SUBTYPE Priority **IS** Any_Priority **RANGE**

Any_Priority'First .. *implementation-defined*;

SUBTYPE Interrupt_Priority **IS** Any_Priority **RANGE**

Priority'Last + 1 .. Any_Priority'Last;

Default_Priority : **CONSTANT** Priority := (Priority'First + Priority'Last) / 2;

An implementation in Ada 95 must support at least 30 values for SUBTYPE Priority and at least one distinct and higher value for SUBTYPE Interrupt_Priority.

An initial priority is assigned to a task by including keyword *pragma* in its declaration:

TASK Stop_Watch **IS**

PRAGMA Priority (20);

END Stop_Watch;

Similarly, a fixed priority can be assigned to all tasks in a task family, if priority pragma is used in a task-type definition:

```
TASK TYPE Soft_Tasking IS
    PRAGMA Priority (Buffer_Priority);
    ENTRY Start_Tasking;
    ENTRY Put_Task_Info (Task_Data : IN Soft_Task_Record);
    ENTRY Get_Task_Info (Task_Data : OUT Soft_Task_Record);
END Soft_Tasking;
```

The base priority of a task is the priority with which it was created. In other words a priority assigned using one of the methods explained above is the base priority of that task. On the other hand, a task also has a second type of priority that is called active priority. The active priority of a task is the same as its base priority unless the task has inherited other priority. Therefore, it would be correct to say that the active priority is the maximum of the task's base priority and any priority it has inherited. The active priority of a task is used to determine the order of task dispatching [ABAW 97].

In Ada 95, the rules describing priority-based scheduling have two parts: the task dispatching model and a specific task dispatching policy. These two rules will be briefly explained through the below sub-subsections.

a. The Task-Dispatching Model

A task becomes a running task only if it is ready to execute and all resources that are needed by that task are available.

The process of selecting a ready task for execution on a processor is known as the *task dispatching*. In Ada 95, the task-dispatching model specifies preemptive scheduling based on conceptual priority-ordered ready queues [ALRM 95].

Conceptual ready queues, task states, and task preemption are the basic factors in specifying *task-dispatching policies*. A ready queue is an ordered list of ready tasks. The first position in a queue is known as the *head of the queue*, and the last position is known as the *tail of the queue*. A task is a ready task only if it is in a ready queue or if it is running. Each processor has one ready queue for each priority level. At any instance, each processor also has a *running task*, which is the task currently being executed by that processor. A preemptible resource such as a processor is a resource that while allocated to one task can be temporarily allocated to another task. A running task is said to be preempted if the processor is allocated to higher-priority task and the running task is placed on the appropriate ready queue.

Task dispatching is done when certain events happen during the execution of a task. The points at which these events happen are called *task-dispatching points*. Whenever a running task on a processor reaches a task dispatching point, the task at the head of the highest priority nonempty ready queue is selected as the new running task and removed from the ready queue. A task dispatching point is reached whenever:

- A task becomes blocked,
- A task becomes ready,
- A task is terminated,
- There is a nonempty ready queue with a higher priority than the priority of the running task,
- The task dispatching policy requires a running task to go back to a ready queue,
- The active priority of a ready task that is not running changes,

- The setting of base priority of a ready task that is not running takes effect,
- The setting of the base priority of a running task takes effect,
- A task executes a delay_statement that does not result in blocking,
- An accept_statement is completed [ALRM 95].

b. The Standard Task Dispatching Policy

Task dispatching policies specify the detailed rules of task dispatching that are not covered by the task-dispatching model. They control when tasks are inserted into and deleted from the ready queues, and whether a task is inserted to the head or the tail of a ready queue. The task dispatching policy is inserted into the program via a Task_Dispatching_Policy pragma.

PRAGMA Task_Dispatching_Policy (*policy_identifier*);

This pragma must be located at the beginning of the first compilation unit (main program) of the partition. The policy identifier can either be *FIFO_Within_Priorities*, which is the only task dispatching policy defined by the language itself, or an implementation-defined identifier. If no such specification is included in any program units comprising a partition, the task dispatching policy for that particular partition stays unspecified and non-deterministic.

When the FIFO_Within_Priorities policy is chosen, the below rules are followed during the modifications to ready queues [ALRM 95]:

- When a blocked task becomes ready, it is added to the tail of the ready queue for its active priority.

- When the active priority of a ready task that is not currently running gets modified, or the setting of its base priority takes effect, the task is taken out from the ready queue for its old active priority and is placed at the tail of the ready queue of its new active priority, except in the case where the active priority is decreased due to the loss of the inherited higher priority, in which case the task is placed at the head of the ready queue for its new active priority.
- When a task is preempted, it is added to the head of the ready queue for its active priority.
- When the setting of the base priority of a running task takes effect, then the task is added to the tail of the ready queue for its active priority.
- When a task executes a delay_statement that does not result in blocking, it is placed at the tail of the ready queue for its active priority.

Although implementations are allowed to define other task dispatching policies, they need not support more than one such policy per partition.

C. ARCHITECTURE OF THE CURRENT CAPS SCHEDULER

In current CAPS architecture, CAPS tools are used in a step-by-step fashion to create a prototype *supervisory program structure* in Ada 95 for the PSDL program defined by the user.

In this step-by-step development, the CAPS translator converts the PSDL program into compilable Ada programs, which include the following five major packages: *exceptions*, *instantiations*, *timers*, *streams*, and *drivers*. All of these packages are preceded by the name of the prototype followed by an underscore.

Exceptions defined by user using PSDL program are contained and instantiated in the exceptions package. Similarly, the instantiations package contains user defined generic package instantiations and the timers package contains user defined timer instantiations. All of the streams used by the prototype are instantiated in the stream package. Streams are implemented as Ada generic protected types embodied in the generic package PSDL_STREAMS, which contains all stream types supported by PSDL. A partial view of the supervisory Ada program for New Thermostat prototype including exceptions, instantiations, timers, and streams packages is illustrated in Figure 4.1.

```
PACKAGE New_Thermostat_Exceptions IS
  -- PSDL exception type declaration
  TYPE PSDL_Exception IS (UNDECLARED_ADA_EXCEPTION);
END New_Thermostat_Exceptions;

PACKAGE New_Thermostat_Instantiations IS
  -- Ada Generic package instantiations
END New_Thermostat_Instantiations;

WITH PSDL_Timers;
PACKAGE New_Thermostat_Timers IS
  -- Timer instantiations
END New_Thermostat_Timers;

WITH New_Thermostat_Exceptions; USE New_Thermostat_Exceptions;
WITH New_Thermostat_Instantiations; USE New_Thermostat_Instantiations;
WITH PSDL_Streams; USE PSDL_Streams;
PACKAGE New_Thermostat_Streams IS

  -- Local stream instantiations
  PACKAGE DS_Hot_Temp_Overheat_Poll IS NEW
    PSDL_Streams.FIFO_BUFFER(Float);

  PACKAGE DS_Cool_Temp_Overcool_Poll IS NEW
    PSDL_Streams.FIFO_BUFFER(Float);

  PACKAGE DS_Take_Temp_Controller IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);

  PACKAGE DS_Heat_Signal_Heater IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Boolean);

  PACKAGE DS_Cool_Signal_Cooler IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Boolean);

-- State stream instantiations

END New_Thermostat_Streams;
```

Figure 4.1. Partial View of Supervisory Program for New_Thermostat Prototype

Current CAPS implementation supports the sampled streams, the state streams, and the data flow streams. Sampled streams are "variables", which data can always be written in and read from. A state stream is a sampled stream with an initial value. Data flow streams are presented as FIFO buffers with size one. Streams in CAPS are implemented as individual Ada protected types with procedures READ, WRITE, and function NEW_DATA.

The other major package created by the translator is the drivers package. The drivers package contains all data declarations. It also embodies data trigger checks that specifies whether or not a stream should be read, the execution trigger checks that governs whether or not an operator should fire, and the output guards that control whether or not an operator should write an output to the output stream. The drivers package also contains calls to procedures implemented by user to simulate the behaviors of atomic operators.

Additional to the packages created by the translator, the CAPS Scheduler generates two task packages, the Static Schedule and the Dynamic Schedule. The static schedule task contains the cyclic schedule for time-critical operators. This cyclic schedule is calculated based on the timing constraints of each time-critical operator entered by the user via PSDL program. The static scheduler task contains calls to the drivers of time-critical operators in a loop. The dynamic schedule task contains calls to the drivers of non-time-critical operators similarly.

Finally, when all these packages created by the translator and the scheduler are combined by one of the CAPS script, they will form the so-called prototype supervisory program. This program is named after the prototype followed by a ".a" extension

standing for an Ada program. CAPS supervisory program structure is shown in Figure 4.2.

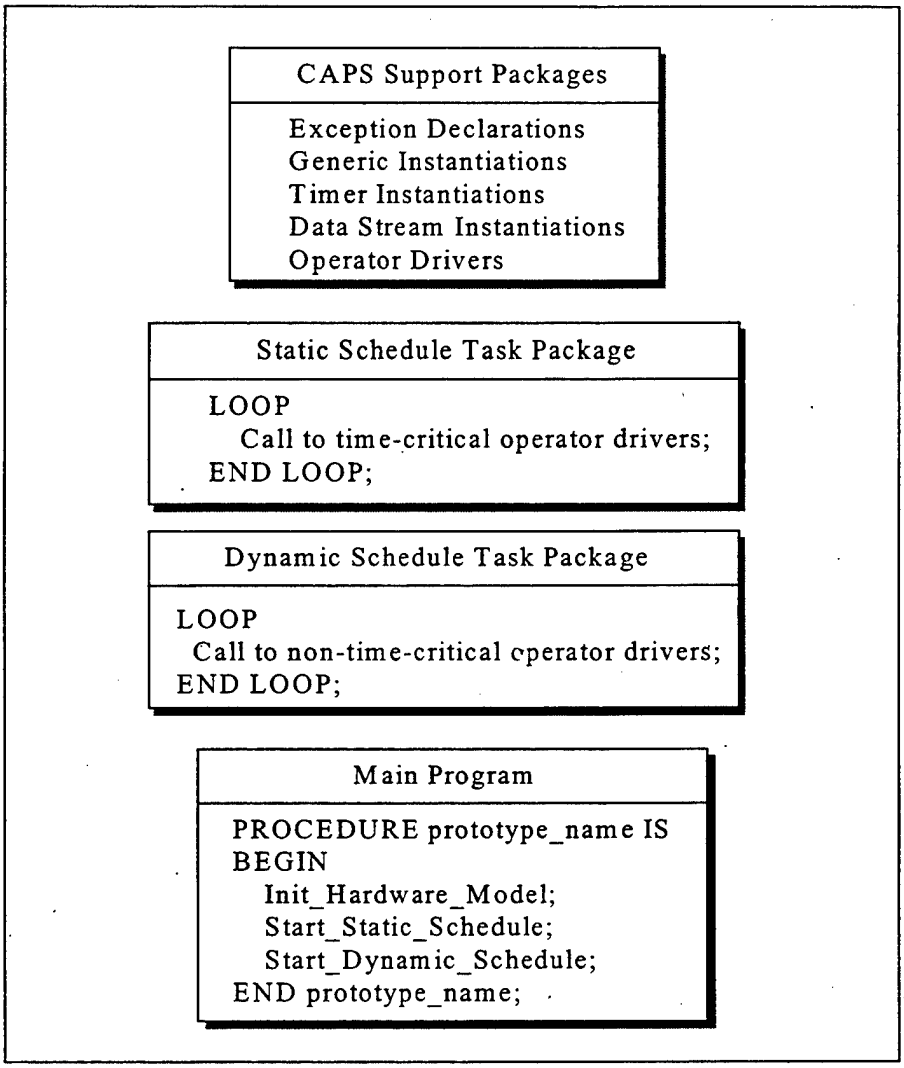


Figure 4.2. CAPS Supervisory Program Structure

The tasks and the protected types in a CAPS generated prototype are associated with one of the priorities in the PRIORITY_DEFINITIONS package:

- *Debugger Task* has the priority four, which is the highest amongst all the tasks and the protected type mentioned. The debugger task deals with the debugging process during the execution of a prototype.

- *Buffer Protected Type* manages all stream implementations with an assigned ceiling priority value of three.
- *Static Scheduler Task* handles all calls to time-critical operators in compliance with the static schedule. The static scheduler task has a priority level of two, therefore all calls made by this task will execute to completion in a non-preemptive way, unless they are preempted by any of debugger tasks or stream implementations. It is worth noting that, even though the buffer protected type has a higher priority level than the static scheduler task, it will not preempt the static scheduler task since the protected operations of the buffers are called by either the static scheduler task or the dynamic scheduler task.
- *Dynamic Scheduler Task* is responsible for calling all non-time-critical operators in a prototype. Non-time-critical operators run according to an execution order defined by the dynamic scheduler, whenever there is a free CPU time left over (slack time) from static schedule. The dynamic scheduler task has the lowest priority value among all other tasks and the buffer protected type, which is one. Therefore, execution of non-time-critical operators can be preempted by any other task type and any stream operation. Even though the dynamic scheduler task has the lowest priority it can introduce priority inversion when it calls the protected buffer type operations. However, this is not a big problem since the protected operations are executed real fast.

D. PROPOSED MULTI-LEVEL CAPS ARCHITECTURE

Current CAPS generated control code deals with tasks that have either hard deadlines or no deadlines at all. In most real-time systems however, there is a third kind

of task with soft deadlines. One of the most common operational areas for tasks having soft deadlines in real-time systems is housekeeping processes. These tasks do not cause serious harm to the system even if they miss their deadlines.

This section of the thesis introduces a new architecture that improves the CAPS generated control code by adding the capability of handling sporadic soft real-time tasks in addition to hard real-time tasks and non-time-critical tasks. Two new major task packages are introduced in the new architecture in order to manage sporadic soft real-time tasks. Modifications are done in some of automatically generated Ada packages by hand. All these new features, modifications, and recently introduced implementation issues will also be discussed in this section. For the sake of simplicity the term of "soft real-time task" will be used to represent the term "sporadic soft real-time task" in the rest of this text.

1. Design Issues of Integrating Soft Real-Time Tasks into CAPS

Adding soft real-time tasks in the current system is an important design issue. Since soft real-time tasks do not have to meet their deadlines all the time, scheduling them using the same criteria for scheduling hard real-time tasks would be very inefficient. Because in this case, the static scheduler would also try to meet deadlines of all soft real-time tasks, which is against the concept of soft deadline. Furthermore, this effort would make the process of finding a feasible schedule much more arduous if not impossible because of the increasing workload on the processor when soft real-time tasks are added in a prototype. Soft real-time tasks, on the other hand, could not be scheduled and executed together with non-time-critical tasks, since they have to complete their execution within deadline once in a while. All these results in the following solution to

the problem of scheduling soft real-time tasks: a new package scheduling soft real-time tasks independent from both hard real-time tasks and non-time-critical tasks. Originated from this idea, three aspects of integrating soft real-time tasks into the current CAPS generated control code are suggested: detection of soft real-time tasks, scheduling of soft real-time tasks, and execution of soft real-time tasks.

a. Detection of Soft Real-Time Tasks

Since the soft real-time tasks are sporadic, a mechanism is needed to detect the arrival time of the new soft real-time tasks. The approach taken to accomplish this job is to create a sporadic server for each soft real-time task. According to this approach, there is one polling task (operator), which is a periodic hard real-time task with relatively short MET corresponding to each soft real-time task (operator). A polling task checks the data and execution triggers of the corresponding soft real-time task and then places it into a task set if the triggering conditions are satisified. This placement is achieved by calling an entry of a new Ada server-tasking package discussed in the following sub-subsection. All these operations are done in the driver of the polling task. Implementation details about polling operator will be discussed in the following section.

b. Scheduling Soft Real-Time Tasks

Next, we need to schedule the soft real-time tasks that are placed into the soft real-time task set. A new Ada server tasking package called SOFT_TASKS_PKG is created to manage the task set. This new package contains entries to place soft real-time tasks into the task set and to retrieve them from the same set. It also assigns deadlines to the soft real-time tasks when they are inserted into the task set. The same package also

schedules the tasks in the set dynamically. The SOFT_TASKS_PKG has the same priority level that the buffer protected type. Scheduling process is repeated whenever a new soft real-time task is inserted into the set and a deadline is assigned to that new task. Control structure and the other implementation issues of SOFT_TASKS_PKG will be covered in the Section E of this chapter.

c. Execution of Soft Real-Time Tasks

Once soft real-time tasks are scheduled by the SOFT_TASKS_PKG, they are ready to execute. For execution of soft real-time tasks, another new Ada tasking package called SRT_EXECUTION_PKG (SRT stands for soft real-time) is introduced to the current architecture. Since CAPS architecture uses the priority based execution order, a priority value lower than static schedule task but higher than dynamic schedule is assigned to this new tasking package. Therefore, soft real-time tasks can be preempted by the debug operations, the stream operations and the hard real-time task execution and can preempt non-time-critical tasks. SRT_EXECUTION_PKG runs whenever there is idle time in the static schedule. It calls the entry of SOFT_TASKS_PKG to retrieve the first task that is scheduled to run by the SOFT_TASKS_PKG and executes the soft real-time task. The way SRT_EXECUTION_PKG is implemented will be explained in the following section.

2. Changes to the Current CAPS Architecture and Modules

Besides the new packages added to the current CAPS code, some other modifications are needed to the current CAPS run-time modules to allow soft real-time tasks, hard real-time tasks and non-time-critical tasks to function together.

a. Modifications to CAPS Supervisory Program Structure

Adding new tasking packages that are explained in the previous sub-section also requires changes in the CAPS Supervisory Program Structure that is covered in Section B of this chapter (Figure 4.2.). In the proposed architecture, CAPS Support Packages, Static Schedule Package and Dynamic Schedule Package are kept unchanged. However, minor changes to the Data Stream Instantiations Package, the Operator Drivers Package, and the Main Program are needed. The most significant modifications to the current program structure are the addition of the SOFT_TASKING_PKG for acquiring and scheduling soft real-time tasks and addition of the SRT_EXECUTION_PKG for execution of scheduled soft real-time. After all these changes, the resultant proposed CAPS Supervisory Program Structure appears as displayed in Figure 4.3. Note that the names of Static and Dynamic Schedule Packages are changed to give a clearer understanding along with a better representation for these packages.

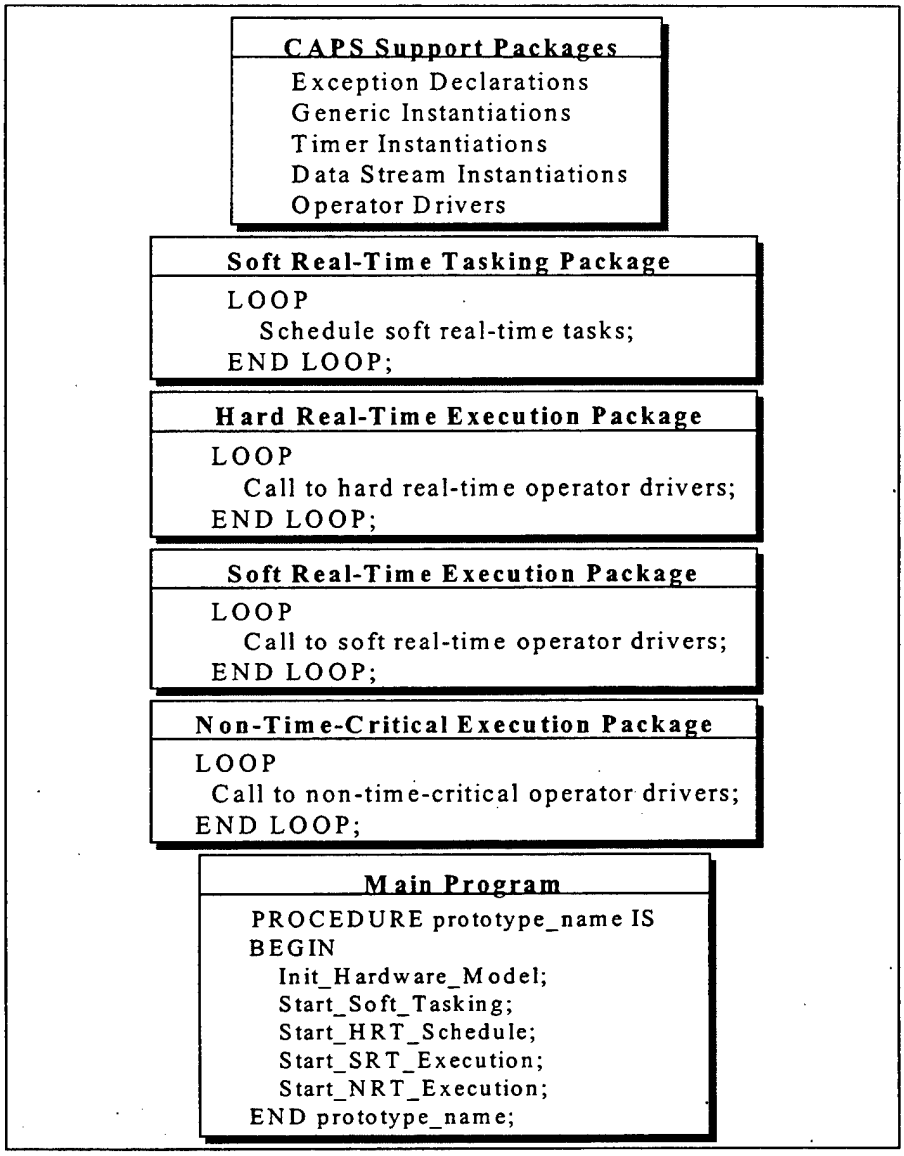


Figure 4.3. Proposed CAPS Supervisory Program Structure

b. Modifications to Main Program

Two new entry calls are required to be added to the driver program to start the new task packages. The current main program and its modified version are displayed in Figure 4.4 to give a more prominent comparison opportunity.

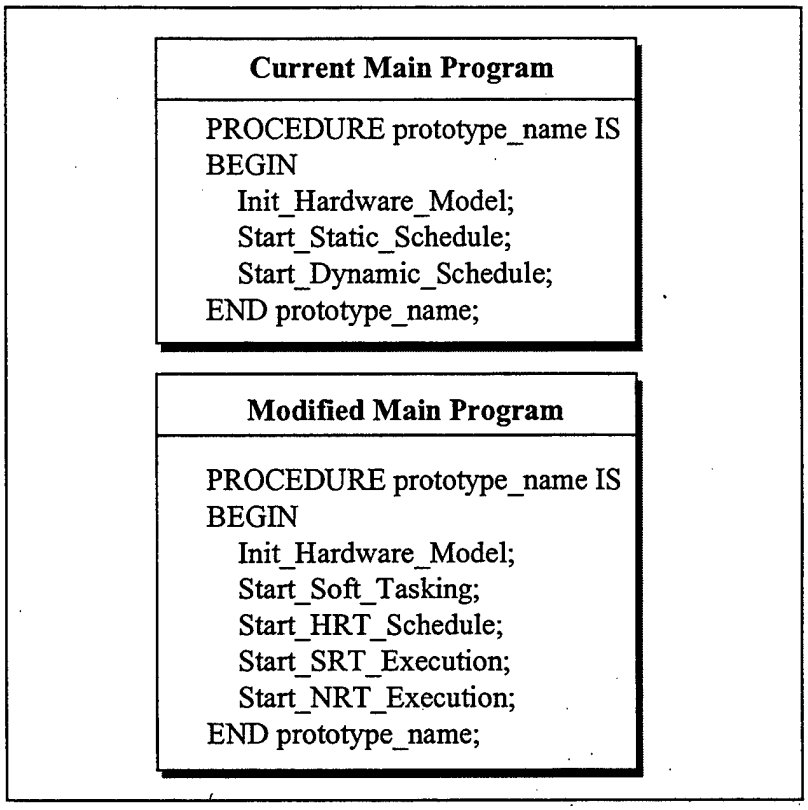


Figure 4.4. The Current and the Modified Main Programs

The order of entry calls in the main program is another important issue. Entry calls must be ordered from the highest priority task to the lowest priority tasks. Otherwise a higher priority task could be blocked by a lower priority task and could never be executed. Accordingly, in the modified main program the call Start_Soft_Tasking comes first since it is the call to a task that has the highest priority among all the other tasks called by rest of the entries. Similarly, Start_NRT_Execution comes last, because the execution of non-time-critical tasks has the lowest priority value.

c. Modifications to PRIORITY_DEFINITIONS Package

Since one more priority level is introduced to the current architecture it is necessary to define the new priority level and adjust the other priority levels accordingly

in the PRIORITY_DEFINITIONS package. The current version of PRIORITY_DEFINITIONS package and the modified version, which is called MOD_PRIORITY_DEFINITIONS package, are illustrated in Figure 4.5.

PRIORITY_DEFINITIONS Package

```
WITH System; USE System;
PACKAGE Priority_Definitions IS

  Debugger_Priority: CONSTANT Priority := 4;
  Buffer_Priority: CONSTANT Priority := 3;
  Static_Schedule_Priority: CONSTANT Priority := 2;
  Dynamic_Schedule_Priority: CONSTANT Priority := 1;

END Priority_Definitions;
```

MOD_PRIORITY_DEFINITIONS Package

```
WITH System; USE System;
PACKAGE Mod_Priority_Definitions IS

   Debugger_Priority: CONSTANT Priority := 5;
   Buffer_Priority: CONSTANT Priority := 4;
   HRT_Schedule_Priority: CONSTANT Priority := 3;
   SRT_Execution_Priority: CONSTANT Priority := 2;
   NRT_Execution: CONSTANT Priority := 1;

END Mod_Priority_Definitions;
```

Figure 4.5. The Current and the Modified PRIORITY_DEFINITIONS Packages

d. Modification to Data Stream Instantiations

Current PSDL graphic editor can only differentiate time-critical operators from non-time-critical operators. There is no way to distinguish hard real-time operators from the soft real-time ones in the existing PSDL specifications yet. However, where the data streams are concerned, there is no need to acknowledge the difference between these

two time-critical task types. Therefore, PSDL graphic editor can be used to insert the data streams required by the soft real-time operators as well.

e. Modifications to Operator Drivers Package

For the same reasons explained in the previous sub-section, the current automated code generation process cannot create the code including driver procedures for soft real-time operators that is needed to implement soft real-time tasks in the drivers package of any prototype. However, the PSDL graphic editor can be used to add a soft real-time operator into a prototype and then be modified by the designer in order to include the soft real-time task properties that is different from hard real-time and non-time-critical task properties. On the other hand, the code fragments that especially synchronize the operations between polling operators and SOFT_TASKS_PKG are hand-patched into the Operator Drivers Package during the implementation phase. Those modifications are covered in detail in the following section in which implementation issues are discussed.

f. Modifications to PSDL_Timers Package

PSDL_Timers Package is another CAPS package that is totally modified throughout this research. Even though this modification is not essential to the proposed multi-level system, it is covered in this sub-subsection because of its advantages in the implementation of the new architecture.

Currently, the CAPS architecture uses *Ada.Calendar Package* as its timing package because when the CAPS was first developed in late 80's, Ada.Calendar was the

only timing package provided by the Ada programming language. However, with Ada 95, another timing package, *Ada.Real_Time*, was introduced.

Calendar is typically "political" time and is not guaranteed to be monotonic since the effects of time zones and daylight saving changes might adjust it. Therefore, Ada.Calendar Package is not suitable to be used in real-time Ada applications in practice. The inclusion of the Real_Time Package in Ada 95 version is based on the realization that there is no other choice to provide a real-time clock which real-time applications could use. The Ada.Real_Tme Package provides the facilities that satisfy the following requirements for a clock that is used to schedule task execution and specify time-outs in a real-time application [ARLL 95]:

- Monotonically non-decreasing time value, incremented at a fixed rate, with bounded discontinuities.
- Fine granularity in time
- The ability to be used as the time reference in all forms of delay statements.
- Efficiency in implementations by using clock facilities that are typical of most existing hardware and real-time operating systems.
- Exact arithmetic on time and duration values, and precise conversion of rational-number duration values to time intervals.
- A defined relationship to other time-related features of the language, including the Ada.Calendar Package, System.Tick, and the Standard.Duration type.

In order to take advantage of the Ada.Real_Time Package in the implementation of the proposed system and to get more precise results in the testing phase of prototypes using this new architecture, PSDL_Timers Package is changed to take advantage of the Ada.Real_Time package. The modified version called *Mod_PSDL_Timers Package* was tested and is ready to use in CAPS right away. The complete code of Mod_PSDL_Timers Package is included in Appendix A.

E. IMPLEMENTATION ISSUES OF PROPOSED ARCHITECTURE

In this section, integrating soft real-time tasks into the current architecture is discussed from the implementation point of view and the same order followed in the discussion regarding to the design issues of integrating soft real-time tasks in the previous section will be followed to keep consistency. Therefore the new concepts will be introduced first. The modifications applied to Data Stream Instantiations and Operator Drivers packages will be covered in details later in the section. To make difficult concepts easier to understand, examples from *New_Thermostat* prototype, which is developed to test this particular proposed multi-level architecture, are addressed throughout the section.

1. Detection and Creation of Soft Real-Time Operators

Detection and creation of soft real-time tasks are realized a series of modifications done in Operator Drivers package. As it is discussed in the previous section there must be one hard real-time polling operator corresponding to each soft real-time operator.

Polling operators are scheduled and executed like all other hard real-time operators. A polling operator fires whenever the time reaches its planned start time

determined by the static scheduler. Once a polling operator fires, first thing it does is to check the data triggers, in other words, it checks to see if there is any new data arrival to the operator. If there is no new data, which means there is definitely no soft real-time task arrived into the system, the operator completes its execution without doing anything. In case of detecting new data in any inbound streams of the TRIGGERED_BY_SOME set[2], the polling operator reads the data from the stream and checks execution triggers. The polling operator then checks the execution guards of corresponding soft real-time operator. Unless execution triggers are satisfied, the polling operator discards the data read and returns. If the execution guards are satisfied, the arrival of a new soft real-time task instance into the system is revealed, but the polling operator must pass one more test before inserting the soft real-time task into the soft real-time task set. In the next step, the polling operator checks if the information in the output data streams from the polling operator to the corresponding soft real-time operator are new or used. If the data are new, which means that they have not been consumed by any soft real-time task yet, then the polling operator just writes the data it has read from the inbound streams into the corresponding outbound streams. The polling operator does not attempt to insert a new soft real-time task instance to the soft real-time task set in this case, because new data in the outbound streams of the polling operator indicate that there is already one soft real-time task instance of that operator ready for execution in the soft real-time task set. The arrival of another instance of the same soft real-time will simply cause the old input to be replaced by the newly arrived data. If the data in the outbound stream are not new, then it

[2] We only consider TRIGGERED_BY_SOME condition for soft real-time tasks since TRIGGERED_BY_ALL condition requires tight compelling between the producer and the consumer, and hence should be handled by hard real-time operators.

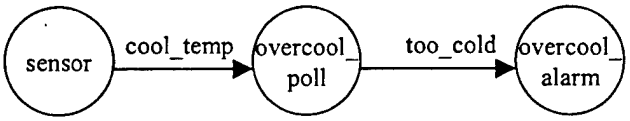
is time to produce a new soft real-time task instance of corresponding soft real-time is time to produce a new soft real-time task instance of corresponding soft real-time operator and insert it into the soft real-time task set. Task attributes, which include the name of the task, MRT specified by the implementation, arrival time of the task and the period of polling operator, are recorded in the newly created task instance. Even though the period of the polling operator seems irrelevant as an attribute of the soft real-time task at this point, it is associated with the task since it will be used to calculate the task instance's deadline in SOFT_TASKS_PKG.

As soon as all required attributes are assigned to the appropriate fields in the task record, communication is established between Operator Drivers Package and the SOFT_TASKING_PKG via an entry call to put the task in the soft real-time task set where ready to execute soft real-time tasks are kept. With this rendezvous, the polling operator wraps up its function of detecting and creating the soft real-time task.

The PSDL graph representation of a polling operator, which is implemented in *New_Thermostat* prototype to detect and create a soft real-time task that displays an alarm message when the temperature decreases below a certain degree, is illustrated in Figure 4.6 along with the code fragments from the driver of the polling operator.

2. Managing and Scheduling Soft Real-Time Tasks

The management of soft real-time tasks in the proposed architecture is handled by a server task implemented in SOFT_TASKS_PKG. This new package defines the server task type to insert all soft real-time tasks in a soft real-time task set, to schedule them in the data structure, and to make them available to execute when the processor is available.




```
-- Data trigger checks.
   IF NOT (DS_Cool_Temp_Overcool_Poll.BUFFER.NEW_DATA) THEN
     RETURN;
   END IF;

 -- Data stream reads.
   DS_Cool_Temp_Overcool_Poll.BUFFER.READ(LV_Cool_Temp);

-- Execution trigger condition check.
   IF ((LV_Cool_Temp < 60.0) AND (EXCEPTION_HAS_OCCURRED = False)) THEN

     Overcool_Poll(LV_Cool_Temp);--Call  to the procedure that only outputs text  for
                                    debugging  purposes.

     IF (DS_Too_Cold_Overcool_Alarm.BUFFER.NEW_DATA) THEN
        DS_Too_Cold_Overcool_Alarm.BUFFER.WRITE(LV_Cool_Temp);
        RETURN;
     ELSE
        DS_Too_Cold_Overcool_Alarm.BUFFER.WRITE(LV_Cool_Temp);
        Overcool_Alarm_Data.Poll_Finish := Host_Duration(Soft_Timer);
        Overcool_Alarm_Data.Task_Name := "Overcool_Alarm";
        Overcool_Alarm_Data.Poll_MET := Target_To_Host(Milliseconds(10));
        Overcool_Alarm_Data.MRT := Target_To_Host(Milliseconds(65));
        Overcool_Alarm_Data.Polling_Period := Target_To_Host(Milliseconds(40));
        Real_Start_Cool := Overcool_Alarm_Data.Arrival_Time; --for monitoring
        Soft_Job.Put_Task_Info(Overcool_Alarm_Data);
     END IF;
   END IF;
```

Figure 4.6. Implementation of Overcool_Poll_Operator and a Part of Its Driver

a. Task Entries

The followings are the three entries associated with this new server task type:

- *Start_Tasking* – This entry is called by the main program to activate the instances of the tasks type.
- *Put_Task_Info* – This entry has three important functions: Placing soft real-time tasks into a task set, assigning a deadline to each soft real-time

task as soon as they are added to the task set, and scheduling them according to an implementation-defined scheduling algorithm.

- *Get_Task_Info* – This entry returns the soft real-time task scheduled to be executed next in the soft real-time task set to the SRT_EXECUTION_PKG.

b. Assigning Deadlines to Soft Real-Time Tasks

A deadline is one of the most important timing constraints for time-critical tasks. Deadlines are assigned to the soft real-time tasks in SOFT_TASKING_PKG in the proposed architecture.

There are three factors effecting deadline calculation of a soft real-time task: The arrival time of the task, the MRT of the task and the period of the corresponding polling operator. With a simplistic approach, one could say that the deadline of a soft real-time is calculated by adding the MRT of the task to the arrival time of the task. However, it would not be correct for the proposed architecture, because the corresponding polling operator introduces a delay in the detection of the arrival of the soft real-time task and takes a small amount of time to complete its own execution after the arrival of the soft real-time task before the soft real-time task can be executed. In other words, the polling operator steals some time from the MRT of the soft real-time task.

In the calculation of the deadline of a soft real-time task the lost time must be subtracted from the MRT of the soft real-time task because it technically cannot be used by the soft real-time task itself. When the lost time is subtracted from the assigned MRT of the task, the actual MRT is obtained. The lost time, in the worst-case, is equal to

the sum of the period and MET of the polling period. Figure 4.7 is provided to help make this concept a little clearer. After all these considerations, the deadline of a soft real-time task is given by the formula:

Deadline = Finishing Time of the Polling Operator + Actual MRT

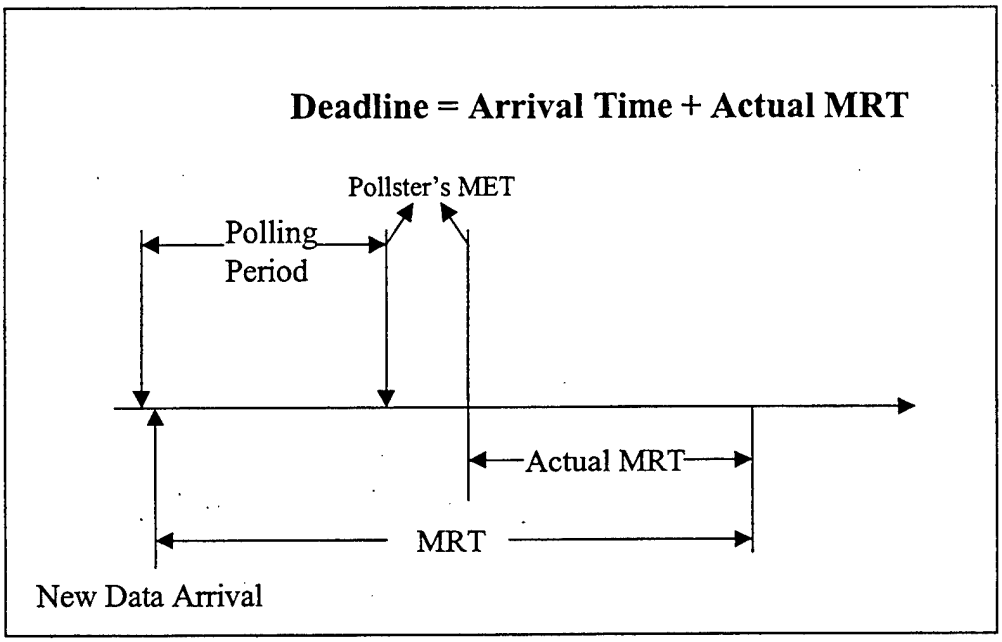


Figure 4.7. Derivation of Soft Real-Time Task Deadline in the Worst-Case

c. Scheduling Soft Real-Time Tasks

Inserting soft real-time tasks into the soft real-time task set and assigning deadlines to them are still not enough to make them ready for execution. If there is more than one task in the task set at a given time, these tasks must be scheduled and put into a certain order before execution.

At this point, a new task instance is added to the task set and a scheduling algorithm should be invoked. Since there is no prior knowledge of the soft real-time task set, the scheduling algorithm has to be a dynamic one. This brings a certain amount of

overhead to the system because the dynamic scheduling algorithm must run each time a new task is added to the task set to insert the new task into the right place according to its timing constraints. Therefore, the overhead caused by the repeated runs of the scheduling algorithm should be carefully taken into consideration during the selection of the algorithm.

In the prototype developed for the experimentation in this research, EDF algorithm is used as the dynamic scheduling algorithm and it works with negligible overhead. However, without excessive empirical data, it would be premature to say that EDF is always the most convenient algorithm for the proposed architecture. Integration of a suitable dynamic algorithm into the proposed architecture for scheduling soft real-time tasks is left as an open research topic for the future work.

The definition of the suggested server task type and overall structure of SOFT_TASKS_PKG are provided in Figure 4.8 to give a general idea about the control statements used to manage soft real-time tasks.

Terminate alternative of select statement in the above code is needed since it provides determinism to the task termination issue. When terminate alternative is used, a task will terminate if all tasks that depend on the same server task have already terminated or are similarly waiting on select statements with terminate alternatives. Terminate alternative provided in Ada language allows server tasks to be constructed that need not concern themselves with termination but will nevertheless terminate when they are no longer needed by other tasks. The lack of this support could cause complicated termination conditions with associated deadlock problems [ABAW 97].

```
--Definition of task type
TASK TYPE Soft_Tasking IS
    PRAGMA Priority (Buffer_Priority);
    ENTRY Start_Tasking;
    ENTRY Put_Task_Info (Task_Data : IN implementation-defined);
    ENTRY Get_Task_Info (Task_Data : OUT implementation-defined);
END Soft_Tasking;

--Instantiation of task instance
Soft_Job : Soft_Tasking;

--Definition of task activation procedure
PROCEDURE Start_Soft_Tasking;

--Body of task Soft_Tasking
TASK BODY Soft_Tasking IS
BEGIN--Task Body Soft_Tasking

    ACCEPT Start_Tasking;

    LOOP
      SELECT
          ACCEPT Get_Task_Info (Task_Data : OUT implementation-defined) DO
             --Return the first task ready to execute to the caller
      OR
          ACCEPT Put_Task_Info (Task_Data : IN implementation-defined) DO
             --Assign deadline to the new task
             --Launch the scheduling algorithm to dynamically schedule tasks
      OR
        TERMINATE;
      END SELECT;
    END LOOP;
  END Soft_Tasking;

--Body of task activation procedure
PROCEDURE Start_Soft_Tasking IS
  BEGIN--Start_Soft_Tasking
    Soft_Job.Start_Tasking;
  END Start_Soft_Tasking;

END Soft_Tasks_Pkg;
```

Figure 4.8. General Overview of Server Package SOFT_TASKS_PKG

3. Execution of Soft Real-Time Tasks

As soon as the processor is idled from executing the hard real-time tasks, the SRT_EXECUTION_PKG, another tasking package introduced in the new architecture to execute soft real-time tasks, is run to execute the soft real-time tasks until it is preempted by a higher priority tasking package.

In order to run a soft real-time task, the SRT_EXECUTION_PKG should retrieve the task to be executed from the soft real-time task set via a rendezvous with the SOFT_TASKS_PKG, where the SOFT_TASKS_PKG passes the first ready task to be executed, together with all task attributes to the SRT_EXECUTION_PKG. The SRT_EXECUTION_PKG checks the task identification and decides which task driver to call to execute the appropriate soft real-time operator, then it calls the corresponding operator driver from Operator Drivers Package and executes the soft real-time task. The definition of the tasks type to execute soft real-time tasks and the overview of SRT_EXECUTION_PKG is shown in Figure 4.9.

```
--Definition of task type
TASK TYPE SRT_Execution_Type IS
   PRAGMA Priority (SRT_Execution_Priority);
   ENTRY Start_Execution;
END SRT_Exceution_Type;
--Instantiation of task
SRT_Execution : SRT_Schedule_Type;
--Definition of the activation procedure
PROCEDURE Start_SRT_Execution;
--Body of task
TASK BODY SRT_Execution_Type IS
BEGIN --SRT_Execution_Type
  ACCEPT Start_Schedule;
  LOOP
      Soft_Job.Get_Task_Info (task_data);
      CASE (task_data.task_id) IS
        WHEN (task_id) =>
          --call appropriate driver
        WHEN (task_id) =>
          --call appropriate driver

      END CASE;
  END LOOP;
END SRT_Execution_Type;
--Body of activation procedure
PROCEDURE Start_SRT_Execution IS
  BEGIN--Start_SRT_Execution
       SRT_Execution.Start_Execution;
END Start_SRT_Execution;
```

Figure 4.9. General Overview of SRT_EXECUTION_PKG

4. Specific Modifications to Existing CAPS Modules

There is no feature to differentiate a hard real-time operator from a soft real-time operator in the existing version of PSDL. In PSDL graph editor, only time-critical and non-time-critical operators can be distinguished from each other. This means that the code associated with soft real-time tasks cannot be automatically generated by the existing CAPS since soft real-time tasks cannot be represented in the PSDL graph and since the PSDL graph is the only specification for the automated code generation. Therefore, in order to implement and test the architecture, some code fragments are added manually to the existing CAPS generated code.

The modifications that are common to all prototypes (e.g. changes made to Main Program and the PRIORITY_DEFINITIONS package) covered in Section D already. Modifications that are prototype-specific (e.g. modifications applied to the Data Stream Instantiations Package and the Operator Drivers Package) are explained by the examples in this sub-section.

a. Operator Drivers Package

The new architecture also requires changes in the Operator Drivers Package. Some of these modifications are done in the poll operator drivers and they are explained in the sub-section in which the detection and the creation details of soft real-time tasks are covered.

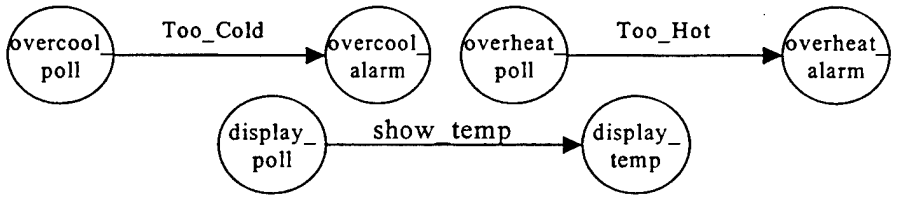
However, there is some more work to be done, because there exist no drivers for soft real-time tasks in the automatically generated Operator Drivers package. Even though they can be inserted into the system via the PSDL graphic editor without the system knowing that they are soft real-time operators, some changes must be done

manually later on. The drivers for the soft real-time tasks should contain only the exceptions and the call to the procedure simulating the behavior of the atomic operator. There is no need to check for execution trigger conditions even though they are sporadic tasks because they are already checked in the poll operator during the detection phase of the task. Similarly, it is not necessary to check the data triggers, because the corresponding poll operator guarantees that the data read from the incoming stream is always written into the data stream going into the soft real-time operator.

b. Insertion of Data Streams to the PSDL Graph

Even though the existing CAPS cannot differentiate a soft real-time operator from a hard real-time task operator or a non-time-critical operator, the PSDL graphic editor can still be used and the data streams of soft real-time operators can be inserted into the prototype automatically as if they are the data streams of a hard real-time or a non-time-critical operator as long as the related operators are in the graph.

In the New_Thermostat prototype, there are two soft real-time operators to display alarm messages when the temperature decreases below a certain degree and increases above a certain degree. Even though there is no data stream coming from these soft real-time operators, there are two data streams going into them from the corresponding poll operators. The graph representation of the soft real-time operators or data streams going into these operators is illustrated in Figure 4.10 together with the modified Data Stream Instantiations package. The piece of code added to the module is intentionally bold-faced to give notice to the reader.




```
WITH New_Thermostat_Exceptions; USE New_Thermostat_Exceptions;
WITH New_Thermostat_Instantiations; USE New_Thermostat_Instantiations;
WITH PSDL_Streams; USE PSDL_Streams;
PACKAGE New_Thermostat_Streams IS
-- Local stream instantiations done automatically by CAPS
  PACKAGE DS_Hot_Temp_Overheat_Poll IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
  PACKAGE DS_Cool_Temp_Overcool_Poll IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
  PACKAGE DS_Take_Temp_Controller IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
  PACKAGE DS_Heat_Signal_Heater IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Boolean);
  PACKAGE DS_Cool_Signal_Cooler IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Boolean);
  PACKAGE DS_Take_Display_Display_Poll IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
-- Stream instantiations related to soft real-time operators (done by hand )
  PACKAGE DS_Too_Hot_Overheat_Alarm IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
  PACKAGE DS_Too_Cold_Overcool_Alarm IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
  PACKAGE DS_Show_Temp_Display_Temp IS NEW
    PSDL_Streams.SAMPLED_BUFFER(Float);
END New_Thermostat_Streams;
```

Figure 4.10. Modified Data Instantiations Package with New Streams

5. Conclusion

When all design and implementation issues addressed in Section D and E are considered, it can be concluded that there are quite a few gaps in the current CAPS architecture to implement the proposed multi-level system. However, only following the directions introduced in Section D and E would be enough to fill these gaps and convert the current architecture of CAPS to the proposed new one. The only part that is not

completely clear yet, is how the dynamic scheduling algorithm for the soft real-time tasks may affect the overall timing behavior of the prototype.

Figure 4.11 presents a pictorial view of the proposed architecture for the new multi-level real-time system and the inter-module communications. In the figure, arrows represent ordinary procedure calls while double arrows represent entry calls (task rendezvous).

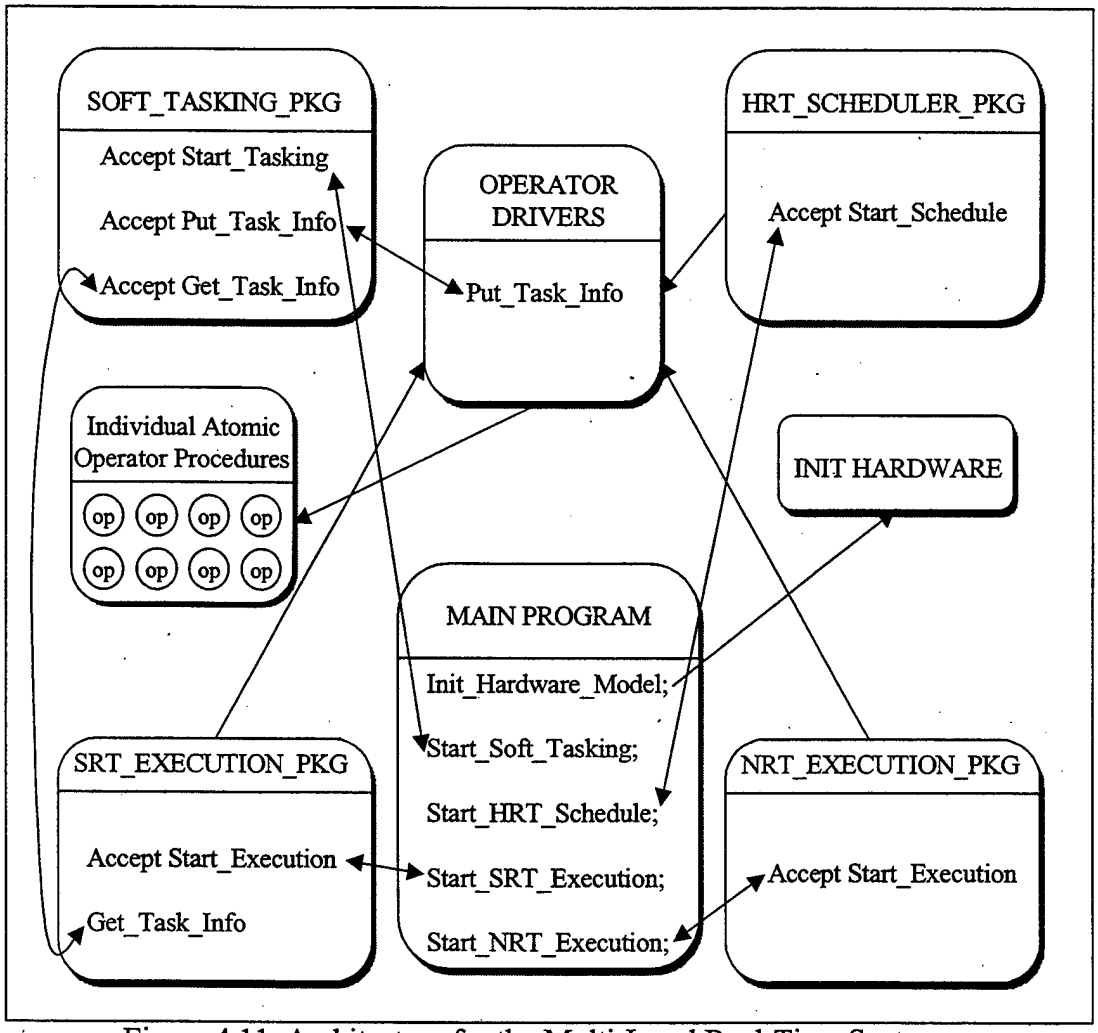


Figure 4.11. Architecture for the Multi-Level Real-Time System

V. EXPERIMENTAL RESULTS

In this chapter, a prototype is developed to examine the behaviors of the proposed multi-level real-time system architecture. Results obtained by running the prototype and the factors affecting the timing requirements are discussed in this chapter.

A. DEVELOPING THE PILOT PROTOTYPE

To test the proposed architecture, a prototype that simulates the behavior of a simple temperature control device is created from an existing sample prototype. A step by step refinement method is followed starting from the original prototype, which contains only hard real-time tasks and non-time-critical tasks to the new one, which contains hard real-time tasks, soft real-time tasks, and non-time-critical tasks.

The original prototype contains 2 time-critical (hard real-time) operators, the sensor operator and the controller operator, and 2 non-time-critical operators, the heater operator and the cooler operator. The sensor operator creates temperature values and sends them to the controller operator. The controller operator checks those values to see if they are below 60°F or above 80°F. If the temperature values are below 60°F, then the controller operator sends a signal to activate the heater operator to bring the temperature up to the acceptable limits. Similarly, if the temperature is above 80°F, the controller operator decides to activate the cooler operator to decrease the temperature. The original *thermostat* prototype and the timing constraints of time-critical operators are shown in Figure 5.1.

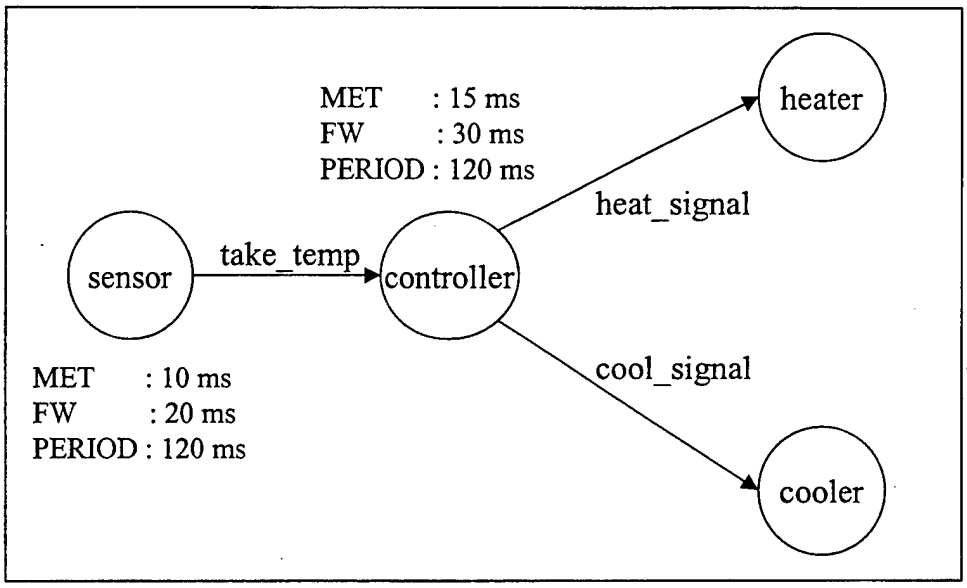


Figure 5.1. The Original *Thermostat Prototype*

After translating and scheduling the above prototype without encountering any problems, the next step is to add three sporadic hard real-time operators into the prototype (Figure 5.2). One of these operators, *display_temp*, displays the temperature values, when they go out of the reasonable limits. The other two, *overheat_alarm* and *overcool_alarm*, display alarm messages when the temperature is above 80°F and is below 60°F, respectively.

Note that the prototype shown in Figure 5.2. does not have a feasible hard real-time schedule because the total load factor of the modified prototype is 1.71. The load factor could be decreased by tightening up METs of operators, but that would cause unwanted timing errors. In other words, operators often would not be able to finish execution within their METs. Assuming that it is acceptable (per user requirements) for the three operators to miss their deadline occasionally, we can obtain a feasible hard real-time schedule by specifying the new operators as soft real-time operators.

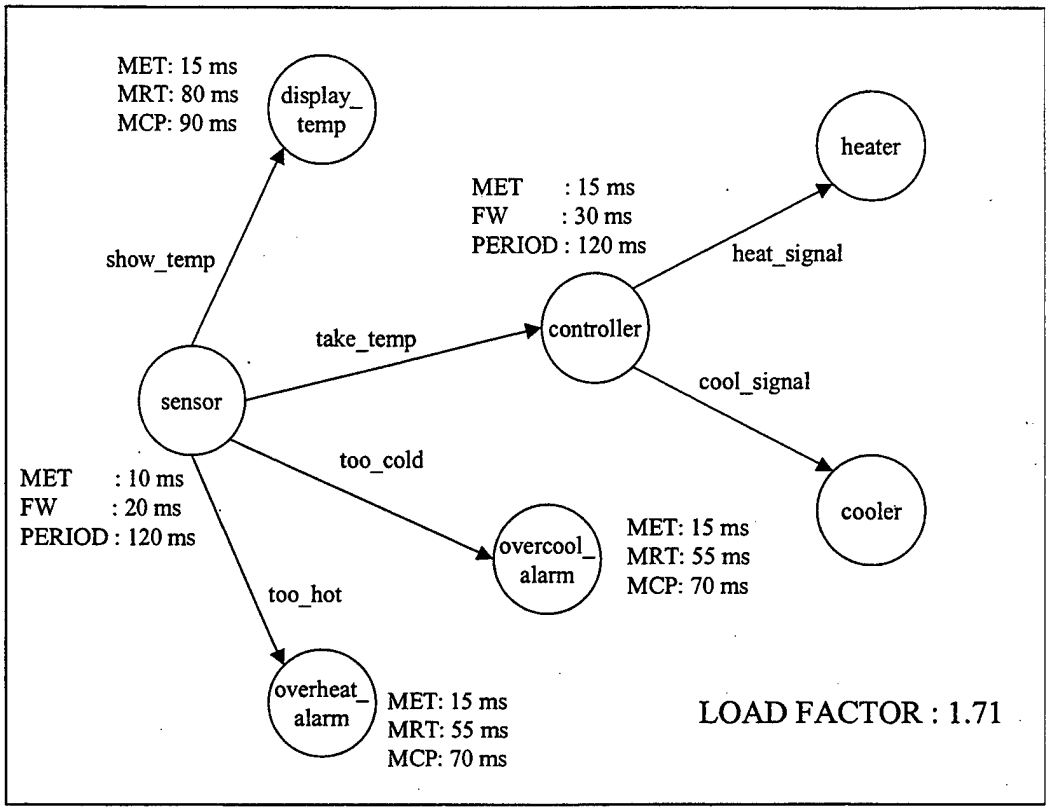


Figure 5.2. The First Version of Modified *Thermostat* Prototype

In order to implement the proposed architecture, three more periodic hard real-time operators to poll three soft real-time sporadic operators are needed. These three polling operators have shorter METs compared to the corresponding soft real-time operators (Figure 5.3).

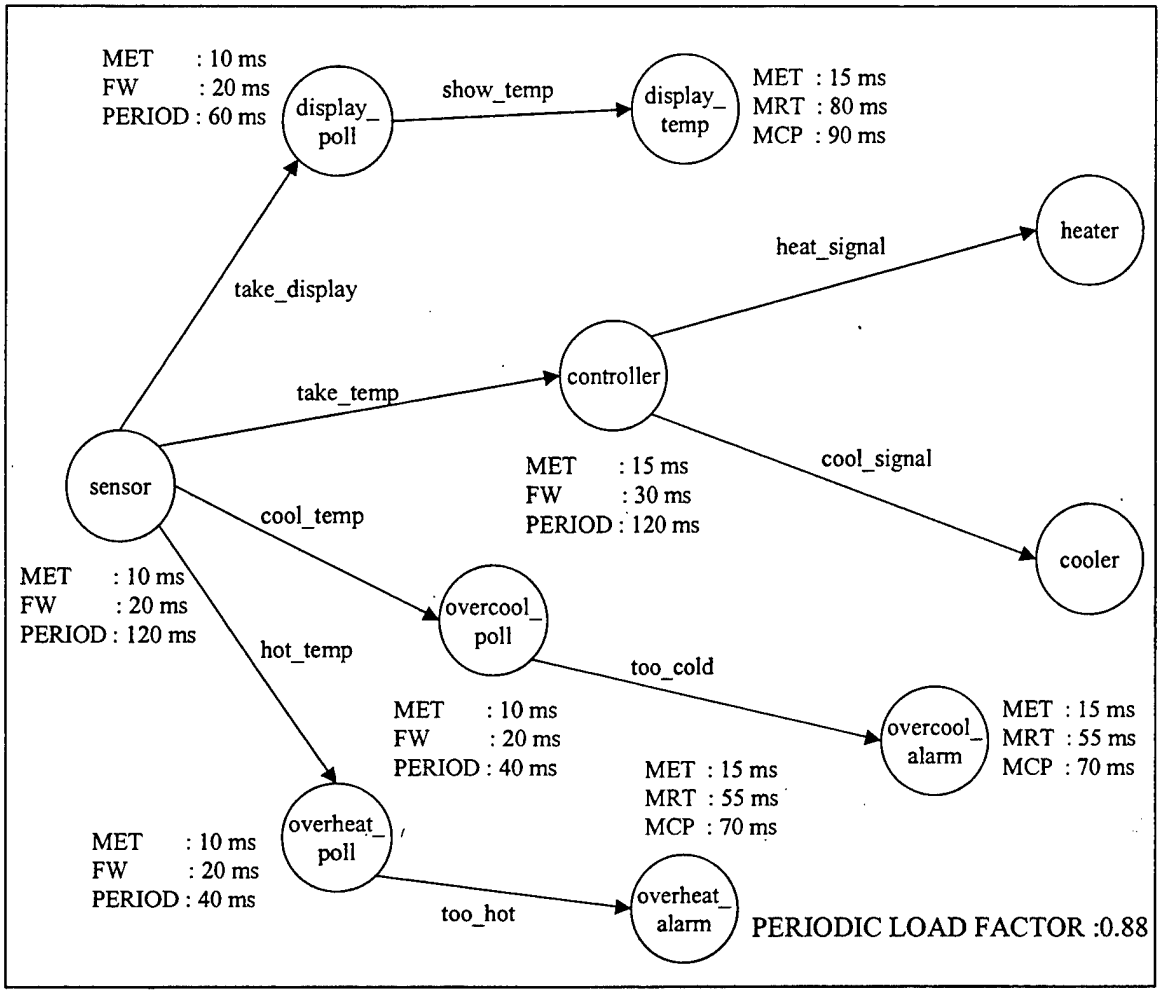


Figure 5.3. New_Thermostat Prototype

Once the CAPS scheduler schedules the hard real-time part of the prototype, which has five hard real-time operators (sensor, controller, display_poll, overheat_poll, and overcool_poll), the rest of the prototype is developed independently from CAPS. The new modules introduced in Chapter IV to handle soft real-time tasks are added to the existing packages and the modules that are automatically generated by CAPS are modified as explained in the previous chapter. Then, the prototype is run repeatedly and the prototype seems to display the expected behavior. However, that is not enough. Since this is a real-time system where meeting timing requirements is as important as producing computationally correct output, a run-time monitoring package is needed to investigate

the behavior of the prototype. The monitoring package used to obtain the statistical information about the prototype execution and the results collected are discussed in the following section..

B. EXECUTION MONITORING AND TEST RESULTS

1. Execution Monitoring System

The developed prototype is run on a simulated clock supported by MOD_PSDL_TIMERS package. In order to monitor the prototype during its execution, the run-time monitoring system developed by Drummond is modified and used to collect timing data [DRUM 97].

One of the changes made to the original monitoring system is to eliminate the RUN_TIME_TIMER module from the system. This module was patterned after the CAPS PSDL_TIMERS to handle all of the creation and administration of real-time timer function. The only difference between RUN_TIME_TIMER package and the CAPS PSDL_TIMERS package was that the timing package which each module uses were different. While the CAPS PSDL_TIMER module uses the ADA.CALENDAR package, the RUN_TIME_TIMER module was designed to use the ADA.REAL_TIME package to obtain better granularity in monitoring. However, PSDL_TIMERS package is modified along the implementation of proposed multi-level architecture to give the whole system a better granularity in time and MOD_PSDL_TIMERS module already uses the ADA.REAL_TIME package. Therefore, RUN_TIME_TIMER module is useless after this modification and is removed from the monitoring system. Removing the RUN_TIME_TIMER module creates an opportunity for combining the two monitoring

packages, RUN_TIME_MEASURE and RUN_TIME_ANALYSIS, into a single RUN_TIME_ANALYSIS module to provide a more compact monitoring system. In the process of combining these two packages the definition of RUN_TIME_RECORD is moved to the RUN_TIME_ANALYSIS module and the code for functions and procedures in the RUN_TIME_MEASURE module is moved to the RUN_TIME_ANALYSIS package.

The other major modification is the definition of the two new timers in the SRT_EXECUTION_PKG and NEW_THERMOSTAT_DRIVERS packages to monitor the execution of soft real-time tasks. The timer defined in SRT_EXECUTION_PKG module monitors the execution of soft real-time tasks in a way similar to the monitoring of the execution of hard real-time tasks by the SCHEDULER_TIMER in HRT_SCHEDULER_PKG module. The timer created in NEW_THERMOSTAT_DRIVERS module is used to timestamp the arrival time of the soft real-time tasks and to timestamp the completion times of soft real-time tasks.

To be able to analyze the execution of soft real-time tasks a new procedure called *Check_Soft_Task_Error* is added to the existing RUN_TIME_ANALYSIS module. This procedure checks if a soft real-time task misses its deadline or not, keeps track of the total number of missed deadlines and the number of consecutive misses, calculates the tardiness for tasks that miss their deadlines, keeps track of the maximum tardiness for each task, and computes the average tardiness for a task.

Changes are also made to the RUN_TIME_RESULTS package to keep up with the modifications done to the other monitoring packages and to make it capable of

displaying the new features of the monitoring package related to soft real-time tasks. The complete code of the modified monitoring system packages is included in Appendix B.

2. Test Results

Since the developed prototype is a multi-level real-time system, more than one performance metric can be used in the evaluation of the system. In other words, different performance metrics can be used to assess the execution of hard real-time tasks and soft real-time tasks.

For hard real-time tasks, the only choice is to meet all deadlines. Otherwise, system would crash. However, *meeting all deadlines* may not be the best suitable metric for soft real-time tasks. If the goal is to meet deadlines of each and every soft real-time task, then it would probably be better to shift those tasks into the hard real-time task category. Metrics such as *minimizing average tardiness, minimizing missed deadlines* or even *minimizing consecutive misses* may be more appropriate for soft real-time tasks.

Of course, to make an assessment, run-time data must be obtained first. Correct statistical data can be collected by running the executable code for a considerably long time. That way, instability caused by momentary hardware failures or uncontrollable sharing problems can be eliminated and more normalized average results can be obtained. Statistical data collected during the execution of the pilot prototype along with the monitoring package is given in Table 5.1 and Table 5.2. Table 5.1 contains test results about hard real-time operators, while Table 5.2 contains test results about soft real-time operators. All issues about the timeliness of the proposed architecture that will be discussed in this sub-section will be based upon the information given in these tables.

Result / Operator	Number of Execution	Planned Run Time	Maximum Run Time	Average Run Time	Number of Missed METs
sensor	1000	10 ms	1.09 ms	0.45 ms	0
controller	1000	15 ms	11.52 ms	2.77 ms	0
display_poll	1000	10 ms	8.68 ms	1.27 ms	0
overcool_poll	1000	10 ms	9.96 ms	0.70 ms	0
overheat_poll	1000	10 ms	20.99 ms	0.80 ms	2

Table 5.1. Test results of Hard Real-Time Operators

Result / Operator	Number of Execution	Planned Run Time	Maximum Run Time	Average Run Time	Number of Missed METs	Number of Missed Deadlines	Maximum Consecutive Misses	Maximum Tardiness	Average Tardiness
display_temp	1000	15 ms	27.22 ms	8.12 ms	1	82	4	211.83 ms	17.12 ms
overcool_alarm	1000	15 ms	112.58 ms	3.91ms	1	129	4	108.94 ms	1.17 ms
overheat_alarm	1000	15 ms	10.39 ms	3.75 ms	0	58	2	75.48 ms	1.89 ms

Table 5.2. Test Results of Soft Real-Time Operators

When the data in Table 5.1 is analyzed, it can be said that the number of timing errors for each hard real-time operator is almost negligible. In other words, almost every execution of each hard real-time task completes in the planned run time. The two over-runs of the maximum execution times are caused by occasional overload of the underlying Unix operating system, because the average run-times of all five operators are quite below the planned run time. Therefore, it would be correct to say that the proposed system meets all the requirements of hard real-time tasks. However, this is expected,

because CAPS was originally designed for this purpose. The data in Table 5.1 tells us that the new architecture maintains this property of CAPS.

Now, it is time to look at the data in Table 5.2 and analyze the results from the soft real-time operators' point of view. First of all, it can be easily seen that the maximum execution times of two of three soft real-time operators are substantially greater than the planned run-times, but on the other hand the average run-times are comparatively less than planned run-time. Therefore, it can reasonably be assumed that extreme run-time values are caused by occasional overload of the underlying Unix operating system. One also can conclude that average run-times of all three soft real-time operators are greater compared to the average run-times of hard real-time operators. This is not an unexpected result either, since execution of soft real-time operators can be preempted by the execution of hard real-time operators, and buffer operations according to priority based scheduling concept used in the proposed architecture.

A closer look at the results shows that the average run-times of two alarm operators, overcool_alarm and overheat_alarm, are very close to each other. This is a sign of the stability of the proposed architecture, because in the test prototype these two operators do the precisely the same job for different soft-real time tasks.

As far as the timing errors caused by soft real-time operators are concerned, the maximum number of missed METs during the execution of a soft real-time operator is only 1 out of 1000 runs, which is acceptable. Furthermore, these timing errors happen when substantially high run-times occur. So, it can be said that those timing errors are also caused by occasional overload of the underlying Unix operating system.

The system does not guarantee that all soft real-time tasks meet their deadlines as it does for hard real-time tasks. Therefore, the number of missed deadlines appears as another important aspect in analyzing soft real-time operator execution. In the test prototype, the maximum percentage of missed deadlines by soft real-time operators is 12.9%, and the average percentage of missed deadlines is 8.9%, which is fairly acceptable. Moreover, the maximum number of consecutive misses is only 4. This is another important issue too. To make this point clearer, an example may be useful. For instance the display_temp operator missed only 82 deadlines out of 1000. Had it missed, for example, 50 deadlines in a row, that result would not be acceptable because in this particular pilot prototype, the temperature might reach unacceptably high limits before it is noticed. For this example, the maximum number of consecutive misses presented in Table 5.2 is also acceptable.

When missed deadlines are considered for soft real-time operators, the amount of tardiness caused by operators is another issue that is worth discussion. One can easily see from Table 5.2 that the maximum and the average tardiness of the display_temp operator are considerably greater than corresponding values of the other two soft real-time operators. When the reason for that is investigated, the roots go to the schedule provided by the CAPS scheduler. When hard real-time operators are entered via the PSDL graphic editor a schedule is found after the translation of the graph and operators are placed into a certain execution order by the CAPS scheduler. According to the PSDL graph entered for the pilot prototype the CAPS scheduler schedule hard real-time operators in the following order with respect to their timing constraints:

SENSOR

OVERCOOL_POLL

OVERHEAT_POLL

DISPLAY_POLL

CONTROLLER

In this order, operator DISPLAY_POLL comes after operator OVERCOOL_POLL and operator OVERHEAT_POLL, therefore operator DISPLAY_TEMP has to be executed after either operator OVERCOOL_ALARM or OVERHEAT_ALARM is finished. The most important determining factor in deadline calculation is the arrival time of the soft real-time tasks. In this case, since alarm pollsters are executed before the display pollster, the alarm operators are placed into the soft real-time task set first and they are scheduled before the display operator by the EDF algorithm. In the proposed architecture soft real-time tasks execute during slack times left from higher priority operations. So, OVERCOOL_ALARM or OVERHEAT_ALARM use the slack time first. Once it is time to execute for DISPLAY_TEMP operator there is less slack time in the system, because most of the slack time is used up by the alarm operator that comes before display operator. Therefore, most of the time DISPLAY_TEMP operators have to wait for some more slack time, causing them to have greater tardiness compared to OVERCOOL_ALARM and OVERHEAT_ALARM operators.

3. Controlling Number of Missed Deadlines and Tardiness

The prototype designer can control both the number of missed deadlines by soft real-time operators and the tardiness of a soft real-time task that misses its deadline. The

key idea in achieving this to adjust the deadline of a soft real-time task, which is computed via the following two formulas as explained in Chapter IV Section E:

Actual MRT = MRT – (Polling Period + MET of Polling Operator) (1)

Deadline = Finishing Time of the Polling Operator + Actual MRT (2)

As can be seen from the formulae, to control the deadline, two factors that take place in deadline calculation may need to be adjusted. The first factor is the MRT of the soft real-time task. The deadline of a soft real-time task depends upon the MRT of the same task. Regardless of the lost time, if the MRT of a soft real-time task gets longer, then the actual_MRT gets bigger as well, according to formula (1). This leads to a less tight deadline in accordance with formula (2). If the MRT entered by the designer is long enough it gives a better laxity to the soft real-time task to execute. In other words, the longer the MRT is, the looser the deadline is.

Since arrival time of a soft real-time task is not controllable, the other factor that can be adjusted by the designer is the lost time. While the designer cannot directly adjust the lost time, the designer can control the lost time indirectly by adjusting the polling period of the pollster. The formula (1) points out that if the polling rate of the pollster gets shorter the time that can be stolen by the pollster from the original MRT of the soft real-time task also gets shorter. This results in a longer actual_MRT and looser deadline for the soft real-time task in accordance with the formula (2).

Even though these two factors can be adjusted to control the deadline of a soft real-time task, the later could cause some consequences. Tightening up the polling period of a pollster would affect the scheduling problem of hard real-time tasks and might result

in an unschedulable prototype. Therefore, it needs more caution to adjust polling period than to adjust the MRT of soft real-time task.

VI. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The real-time computing or even the hard real-time computing and scheduling problem is not completely solved yet. In this immense research area, this thesis opens another window and tries to explore a new real-time system architecture that comprises the properties of both hard and soft real-time systems without making concessions from the hard real-time part, in other words meeting all hard real-time constraints.

Soft real-time tasks are executed in the slack time between the execution of the hard real-time tasks in the proposed architecture, as they are in most similar multi-level real-time systems. However, since the detection of soft real-time tasks is handled by corresponding hard real-time tasks, and since the managing and dynamic scheduling of soft real-time tasks are achieved by using a higher priority task than hard real-time scheduler task, soft real-time tasks are prevented from being starved for scheduling time. Furthermore, the results presented in Chapter V shows that a very acceptable utilization is realized for soft real-time tasks.

Throughout the implementation of the proposed architecture, required concurrent programming is realized by making use of the power of the Ada 95 tasking concept. The hierarchy created among different tasks by assigning correct priorities to correct tasks also proves that priority assignment is an important issue in real-time scheduling and computing and it is useful when it is done correctly.

A method for assigning deadlines dynamically to soft real-time tasks is developed and implemented in the new multi-level architecture. The scheduling of soft real-time

tasks are also achieved dynamically via the EDF algorithm. Using these two procedures, the proposed system gains the capability of responding to unpredictable stimuli in a predictable manner and the capability of reacting to all possible events in a timely manner.

The run-time monitoring package, which is improved by adding features to gather run-time information about soft real-time tasks helps analyze the performance of the system in depth. More than one performance metric can be used to evaluate the system.

B. SUGGESTED CAPS MODIFICATIONS

Although the proposed system is built and tested outside the CAPS environment, modules created by CAPS tools are used as the core modules for the new architecture. Hence, the proposed architecture can be integrated into CAPS environment after a series of modifications to CAPS is realized.

1. Modification to PSDL Specifications

To integrate the proposed system into CAPS, the first thing to do is define the sporadic soft real-time task concept in the PSDL specifications so that CAPS can differentiate soft real-time tasks from hard real-time tasks and from non-time-critical tasks. Along with this modification, some changes are needed to the PSDL graphic editor to make the designer capable of entering soft real-time tasks and their timing constraints via the editor together with hard real-time tasks.

2. Automated Code Generation for Soft Real-Time Tasks

CAPS can be modified so as to generate code for soft real-time tasks automatically as it does for hard real-time tasks. For this purpose the polling operators

and the associated output streams should be automatically added to the PSDL graphs by the scheduler. Data stream instantiations coming from or going into soft real-time tasks can be included in the Data Stream Instantiations package, and driver procedures for soft real-time tasks can be contained in the Operator Drivers package. While changes are being made in automated code generation, including triggering conditions of soft real-time operators in corresponding polling operator drivers may require close attention. Moreover, the skeletons of Soft Real-Time Tasking and Soft Real-Time Execution Packages can be generated automatically after translation of the prototype graph. In parallel with the code fragments generated for soft real-time tasks in the above-mentioned modules, code coping with soft real-time execution and scheduling should be automatically included in the main program also.

3. Modifications to Certain CAPS Modules

In order to make the CAPS system compatible with the proposed architecture, modifications to PRIORITY_DEFINITIONS package and PSDL_TIMERS package should be realized as they are explained in Chapter IV, Section D as well.

C. FUTURE RESEARCH RECOMMENDATIONS

1. Periodic and Sporadic Soft Real-Time Tasks

Note that this research only considers sporadic soft real-time tasks as the soft real-time part of the multi-level real-time system. However, periodic and aperiodic real-time tasks also may have soft deadlines. Further research may be needed to include periodic and aperiodic soft real-time tasks along with sporadic soft real-time tasks in the soft real-time part of a more improved multi-level system.

2. Trying Different Scheduling Algorithms

Due to the difficulty of building a new architecture, the first and the only scheduling algorithm used in the scope of this thesis for scheduling soft real-time dynamically is the well known Earliest Deadline First (EDF) algorithm. However, other scheduling algorithms may be more appropriate for scheduling soft real-time tasks in such a multi-level real-time system. Candidate algorithms resulting from a more exhaustive literature search should be tested that could be used in the proposed architecture. Then, the best candidate that brings the least overhead to the system might be selected as the permanent scheduling algorithm in the new architecture, or a selection could be provided to users.

3. Soft Real-Time and Hard Real-Time Scheduling Interactions

In the proposed system, hard real-time tasks are scheduled statically before the execution of a prototype is started, while soft real-time tasks are scheduled dynamically independent from hard real-time tasks during the execution of the prototype. As a result of this implementation choice, the load put on the single processor by hard real-time tasks is presented by the CAPS scheduler right after the scheduling process of hard real-time operators is completed, but the load brought by soft real-time tasks stays hidden even when the execution of the prototype is terminated. Since this research did not investigate the load issue caused by soft real-time operators, this topic might be covered in detail in another research with the effects of high and low soft real-time tasks to the behavior and the timeliness of the system.

LIST OF REFERENCES

[ABAW 97] Burns,A. and Wellings, A.J., *Real-Time Systems and Programming Languages*, Addison Wesley, 1997.

[ALRM 95] Ada 9X Mapping/Revision Team, *Ada 95, The Language Reference Manual & Standart Libraries*, Intermetrics, Inc., January 1995.

[ARLL 95] Ada 9X Design Team, *Ada 95, Rationale, The Language & Standard Libraries*, Intermetrics, Inc., January 1995.

[BDTI 93] Burns, A., Davis, R.I., and Tindell, K.W., *Scheduling Slack Time in Fixed Priority Preemptive Systems*, Proceedings of Real-Time Systems Symposium, pp. 221-227, December 1993.

[BURN 91] Burns, A., *Scheduling Real-Time Systems: A Review*, Software Engineering Journal, Vol 6, No 3, pp116-128, 1991.

[CHET 89] Chetto, H. and Chetto, M., *Some Results of the Earliest Deadline Scheduling Algorithm*, IEEE Transactions on Software Engineering, Vol. 15, no. 10, pp. 1261-1269, 1989.

[CORD 95] Cordeiro, M.M., *Distributed Hard Real-Time Scheduling for a Software Prototyping Environment*, Ph.D. Dissertation, Naval Postgraduate School, Monterey, CA, March 1995.

[JENS 98] Jensen, E.D., *Scheduling in Real-Time Systems*, http://www.realtime/-os.com/sched_o3.html, September 1998

[LERT 92] Lehoczky, J.P. and Ramos-Thuel, S., *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems*, Real-Time Systems Symposium, pp. 110-124, 1992.

[LESD 87] Lehoczky, J.P., Sha, L., and Ding, V., *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior*, Technical Report, Department of Statistics, Carnegie-Mellon University, 1987.

[LIHE 96] Lin, K. and Herkert, A., *Jitter Control in Time-Triggered Systems,* Department of Electrical and Computer Engineering, University of California, Irvine, 1996.

[LILA 73] Liu, C.L. and Layland, J.W., *Scheduling Algorithms for Multiprogramming in a Hard Real_Time Environment*, Journal of the ACM, vol. 20, no. 1, pp. 46-61, 1973.

[LUBY 88] Luqi, Berzins, V., and Yeh, R.T., *A Prototyping Language for Real-Time Software*, IEEE Transactions on Software Engineering, vol. 14, no.10, pp. 1409-1423, October 1988.

[LUQI 89] Luqi, *Software Evolution Through Rapid Prototyping*, IEEE Computer, pp. 13-25, May 1989.

[LUSH 96] Luqi and Shing, M., *Real-Time Scheduling for Software Prototyping*, Journal of Systems Integration, pp. 41-72, 1996.

[RAST 94] Ramamritham, K. and Stankovic, J.A., *Scheduling Algorithms and Operating Systems Support for Real-Time Systems*, Proceedings of the IEEE, vol. 82, no. 1, pp. 55-67, January 1994.

[SRLE 90] Sha, L., Rajkumar, R., and Lehoczky, J.P., *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, IEEE Transactions on Computers, 1990.

[SSDB 95] Stankovic, J.A., Spuri, M., Di Natale, M., and Buttazzo, G., *Implications of Classical Scheduling Results for Real-Time Systems*, pp. 1-15, IEEE Computer, vol. 28, no. 6, pp. 16-25, June 1995.

[STAN 96] Stankovic, J.A., *Real-Time and Embedded Systems*, ACM Computing Surveys, vol. 28, no. 2, pp. 205-208, March 1996.

[TALI 94] Tarng, W. and Lin, T., *Characterization of Scheduling Algorithms in Real-Time Computing Systems*, Proceedings of 1994 IEEE Region 10's Ninth Annual International Conference, vol.2, pp.617-622, August 1994.

[TIMM 98] Timmerman, M., *Real-Time Encyclopedia*, www.real-time-info.be info.be/encyc/techno/terms/defini/def.htm, September 1998.

APPENDIX A -MOD_PSDL_TIMERS PACKAGE

```
WITH Ada.Real_Time; USE Ada.Real_Time;
PACKAGE Mod_PSDL_Timers IS
-------------------------------------------------------------------------------
-- FILE          : Mod_PSDL_Timers                                            --
-- DATE          : 28 APR 98                                                  --
-- MODIFIED BY   : LTJG Omer Korkut                                           --
-- COMPILER      : Sun/Ada                                                    --
-- DESCRIPTION   : This package is modified from original PSDL_Timers         --
--                 by including Ada.Real_Time timing package instead          --
--                 former Calendar timing package.                            --
-------------------------------------------------------------------------------

  SUBTYPE Millisec IS Natural;
  TYPE Timer IS PRIVATE;

  -- Operations invoked from PSDL control constraints.

  -- Timer reading wrt the target machine.
  FUNCTION Read(Name: IN Timer) RETURN Millisec;
  PROCEDURE Reset(Name : Timer);
  PROCEDURE Start(Name : Timer);
  PROCEDURE Stop (Name : Timer);

  -- Operations used by the CAPS tools.

  -- Creates and initializes a new timer.
  FUNCTION New_Timer RETURN Timer;

  -- Total time accumulated in the Timer on the host machine.
  FUNCTION Host_Duration(Name : Timer) RETURN Time_Span;

  -- Converts durations on the target machine to the
  -- corresponding durations on the CAPS host machine.
  FUNCTION Target_To_Host(TS: Time_Span) RETURN Time_Span;

  PROCEDURE Stop_All_Timers;
  PROCEDURE Start_All_Timers;
  PROCEDURE Reset_All_Timers;
-- Subtract T (host machine duration) from the reading on the timer
  PROCEDURE Subtract_Host_Time(TS : Time_Span; Name : Timer);

  -- Subtract T (host machine duration) from the reading on all timers
  PROCEDURE Subtract_Host_Time_From_All_Timers(TS : Time_Span);
```

```
   PRAGMA Inline(Read, Reset, Start, Stop);
   PRAGMA Inline(Host_Duration, Stop_All_Timers, Start_All_Timers);

   PRIVATE
   TYPE State IS (Running, Stopped);
   TYPE Timer_Record IS
     RECORD
     -- All times in a Timer_Record are wrt the caps host machine.
     Start_Time    : Time; -- Meaningful only if Present_State = Running.
     Elapsed_Time  : Time_Span := Time_Span_Zero;
     Present_State : State := Stopped;
   END RECORD;

   TYPE Timer IS ACCESS Timer_Record;

END Mod_PSDL_Timers;


WITH CAPS_Hardware_Model; USE CAPS_Hardware_Model;
WITH PSDL_Timer_Lists; USE PSDL_Timer_Lists;
PACKAGE BODY Mod_PSDL_Timers IS
-------------------------------------------------------------------------------
-- FILE             : Mod_PSDL_Timers                                        --
-- DATE             : 28 APR 98                                              --
-- MODIFIED BY      : LTJG Omer Korkut                                       --
-- COMPILER         : Sun/Ada                                                --
-- DESCRIPTION      : This package is modified from original PSDL_Timers     --
--                    by including Ada.Real_Time timing package instead      --
--                    former Calendar timing package.                        --
-------------------------------------------------------------------------------

   -- A list containing all the timers in the prototype and schedulers
   Timers : Timer_List := Empty_Timer_List;

   -- Converts elapsed time to milliseconds
   FUNCTION Convert_To_Target_Time(TS : Time_Span) RETURN Millisec IS

     Conversion_Factor : CONSTANT Float := 1000.0;
     Dur : Duration := 0.0;

   BEGIN--Convert_To_Target_Time

     Dur := To_Duration(TS);
     RETURN Millisec(Float(Dur) * Conversion_Factor / CPU_Speed_Ratio);

   END Convert_To_Target_Time;
```

```
-- Converts time periods on the target machine to the corresponding
-- time periods on the CAPS host machine.
FUNCTION Target_To_Host(TS : Time_Span) RETURN Time_Span IS

  Dur    : Duration := 0.0;
  Result : Time_Span;

BEGIN--Target_To_Host

  Dur := To_Duration(TS);
  Dur := Duration(Float(Dur) * CPU_Speed_Ratio);
  Result := To_Time_Span(Dur);
  RETURN(Result);

END Target_To_Host;

FUNCTION Read(Name : IN Timer) RETURN Millisec IS

  Inter_Time : Time;

BEGIN--Read

  CASE Name.Present_State IS
    WHEN Running =>
      Inter_Time := Name.Elapsed_Time + Clock;
      RETURN(Convert_To_Target_Time(Inter_Time - Name.Start_Time));
    WHEN Stopped =>
      RETURN(Convert_To_Target_Time(Name.Elapsed_Time));
  END CASE;

END Read;

PROCEDURE Reset(Name : Timer) IS
BEGIN--Reset

  Name.Elapsed_Time := Time_Span_Zero;
  CASE Name.Present_State IS
    WHEN Running => Name.Start_Time := Clock;
    WHEN Stopped => Null;
  END CASE;

END Reset;

PROCEDURE Start(Name : Timer) IS
BEGIN--Start
```

```
  CASE Name.Present_State IS
    WHEN Running => null;
    WHEN Stopped =>
      Name.Present_State := Running;
      Name.Start_Time := Clock;
  END CASE;

END Start;

PROCEDURE Stop(Name : Timer) IS

  Inter_Time : Time;

BEGIN--Stop

  CASE Name.Present_State IS
    WHEN Running =>
      Name.Present_State := Stopped;
      Inter_Time := Name.Elapsed_Time + Clock;
      Name.Elapsed_Time := Inter_Time - Name.Start_Time;
    WHEN Stopped => null;
  END CASE;

END Stop;

-- Creates and initializes a new timer
FUNCTION New_Timer RETURN Timer IS

  Result : Timer;

BEGIN--New_Timer

  Result := NEW Timer_Record;
  Add(Result, Timers);
  RETURN Result;

END New_Timer;

-- Total time accumulated in the Timer on the host amchine, used
-- instead of the real-time clock in the static schedule
FUNCTION Host_Duration(Name : Timer) RETURN Time_Span IS

  Inter_Time : Time;

BEGIN--Host_Duration

  CASE Name.Present_State IS
```

```
      WHEN Running =>
        Inter_Time := Name.Elapsed_Time + Clock;
        RETURN(Inter_Time - Name.Start_Time);
      WHEN Stopped => RETURN(Name.Elapsed_Time);
    END CASE;

  END Host_Duration;

  PROCEDURE Stop_All_Timers IS

    TList : Timer_List := Timers;

  BEGIN--Stop_All_Timers

    WHILE TList /= Empty_Timer_List LOOP
      Stop(First(TList));
      TList := Rest(TList);
    END LOOP;

  END Stop_All_Timers;

PROCEDURE Start_All_Timers IS

    TList : Timer_List := Timers;

  BEGIN--Stop_All_Timers

    WHILE TList /= Empty_Timer_List LOOP
      Start(First(TList));
      TList := Rest(TList);
    END LOOP;

  END Start_All_Timers;

PROCEDURE Reset_All_Timers IS

    TList : Timer_List := Timers;

  BEGIN--Reset_All_Timers

    WHILE TList /= Empty_Timer_List LOOP
      Reset(First(TList));
      TList := Rest(TList);
    END LOOP;

  END Reset_All_Timers;
```

```
  -- Subtract TS (Host Machine Duration) from the reading on timer Name
  PROCEDURE Subtract_Host_Time(TS : Time_Span; Name : Timer) IS
  BEGIN--Subtract_Host_Time
    Name.Elapsed_Time := Name.Elapsed_Time - TS;
  END Subtract_Host_Time;

  -- Subtract TS (Host Machine Duration) from the reading on all timers
  PROCEDURE Subtract_Host_Time_From_All_Timers(TS : Time_Span) IS

    TList : Timer_List := Timers;

  BEGIN--Subtract_Host_Time_From_All_Timers

    WHILE TList /= Empty_Timer_List LOOP
      Subtract_Host_Time(TS,First(TList));
      TList := Rest(TList);
    END LOOP;

  END Subtract_Host_Time_From_All_Timers;

END Mod_PSDL_Timers;
```

APPENDIX B – RUN-TIME MONITORING PACKAGE

```
WITH Ada.Real_Time; USE Ada.Real_Time;
WITH Ada.Text_IO; USE Ada.Text_IO;
PACKAGE Run_Time_Analysis IS
-------------------------------------------------------------------------------- --
-- FILE             : Run_Time_Analysis.ads                                     --
-- DATE             : 29 JUL 98                                                 --
-- MODIFIED BY      : LTJG Omer Korkut                                          --
-- COMPILER         : Sun/Ada                                                   --
-- DESCRIPTION      : This module is called to perform an analysis upon the     --
--                    CAPS execution run time.                                  --
-- MODIFICATION     : The package is modified to be able to handle the run      --
--                    time analysis of the operators that executes more than    --
--                    once in a single period.                                  --
-------------------------------------------------------------------------------- --

  Name_Size     : CONSTANT Integer := 23;
  Op_Type_Size : CONSTANT Integer := 4;

  -- Record of operator data
  -- All of the RECORD elements are per instance of each operator.

  TYPE Run_Time_Record IS
    RECORD
      Operator_Name       : String(1..Name_Size);
      Operator_Type       : String(1..Op_Type_Size) := "Hard";
      Arrival_Time        : Time_Span := Time_Span_Zero;
      Execution_Start     : Time_Span := Time_Span_Zero;
      Execution_Stop      : Time_Span := Time_Span_Zero;
      Planned_Start       : Time_Span := Time_Span_Zero;
      Planned_Stop        : Time_Span := Time_Span_Zero;
      Actual_Run_Time     : Time_Span := Time_Span_Zero;
      Planned_Run_Time    : Time_Span := Time_Span_Zero;
      Overhead            : Time_Span := Time_Span_Zero;
      Response_Time       : Time_Span := Time_Span_Zero;
      Completion_Time     : Time_Span := Time_Span_Zero;
      Total_Run_Time      : Time_Span := Time_Span_Zero;
      Deadline            : Time_Span := Time_Span_Zero;
      Total_Timing_Error  : Integer := 0;
      Average_Timing_Error: Time_Span := Time_Span_Zero;
      Average_Run_Time    : Time_Span := Time_Span_Zero;
      Minimum_Run_Time    : Float := 100.0;
      Maximum_Run_Time    : Time_Span := Time_Span_Zero;
      Execution_Count     : Integer := 0;
```

```
      Run_Time_Difference    : Time_Span := Time_Span_Zero;
      Total_Missed_Deadlines : Integer := 0;
      Individual_Tardiness   : Time_Span := Time_Span_Zero;
      Total_Tardiness        : Time_Span := Time_Span_Zero;
      Maximum_Tardiness      : Time_Span := Time_Span_Zero;
      Average_Tardiness      : Time_Span := Time_Span_Zero;
      Consecutive_Misses     : Integer := 0;
      Max_Consecutive_Misses: Integer := 0;
      Miss_Flag              : Boolean := False;
   END RECORD;

   TYPE Run_Time_Array IS ARRAY (Integer RANGE <>) OF Run_Time_Record;

   -- Run Time Analysis procedures declaration
   PROCEDURE Analyze_Operator_Execution_Data
         (Result_Data : IN OUT Run_Time_Record;
          Operator_Data: IN OUT Run_Time_Array);

   PROCEDURE Get_Operator_Timing_Data(Result_Data : IN OUT
                                      Run_Time_Record;
                                      Operator_Data : IN OUT
                                      Run_Time_Record);

   PROCEDURE Analyze_Results_Data(Result_Data : IN OUT Run_Time_Record;
                                  Operator_Data : IN OUT Run_Time_Record);

   PROCEDURE Check_Soft_Task_Error(Result_Data : IN OUT Run_Time_Record;
                                   Operator_Data : IN OUT Run_Time_Record);

   PROCEDURE Send_Operator_Run_Time_Calculation_Data
                  (Result_Data:IN OUT Run_Time_Record);
END Run_Time_Analysis;
```

```
WITH Run_Time_Results; USE Run_Time_Results;
WITH Soft_Tasks_Pkg; USE Soft_Tasks_Pkg;
WITH Ada.Real_Time; USE Ada.Real_Time;
WITH Ada.Float_Text_IO; USE Ada.Float_Text_IO;
WITH Ada.Integer_Text_IO; USE Ada.Integer_Text_IO;
PACKAGE BODY Run_Time_Analysis IS
-------------------------------------------------------------------------------- --
-- FILE              : Run_Time_Analysis.adb                                    --
-- DATE              : 29 JUL 98                                                --
-- MODIFIED BY       : LTJG Omer Korkut                                         --
-- COMPILER          : Sun/Ada                                                  --
-- DESCRIPTION       : This module is called to perform an analysis upon the    --
--                     CAPS execution run time.                                 --
-- MODIFICATION      : The package is modified to be able to handle the run     --
--                      time analysis of the operators that executes more than  --
--                      once in a single period.                                --
-------------------------------------------------------------------------------- --

--*************************************************************--
-- Analyze_Operator_Execution_Data                              --
--                                                              --
-- This procedure is called by the scheduler program to get actual run    --
-- time data from Run_Time_Measure module. This run time data includes    --
-- actual execution starting and stopping time of the CAPS operator.      --
--*************************************************************--
  PROCEDURE Analyze_Operator_Execution_Data
              (Result_Data  : IN OUT Run_Time_Record;
               Operator_Data: IN OUT Run_Time_Array)  IS

  BEGIN--Analyze_Operator_Execution_Data

    FOR I IN 1 .. Operator_Data'Last LOOP

      IF (Operator_Data(I).Execution_Stop >
            Operator_Data(I).Execution_Start) THEN

        -- Calculate Actual Run Time
        Operator_Data(I).Actual_Run_Time :=
                    Operator_Data(I).Execution_Stop -
                    Operator_Data(I).Execution_Start;


        Result_Data.Actual_Run_Time := Operator_Data(I).Actual_Run_Time;

        Result_Data.Execution_Count := Result_Data.Execution_Count + 1;
```

```
      Result_Data.Execution_Start := Operator_Data(I).Execution_Start;

      Result_Data.Execution_Stop  := Operator_Data(I).Execution_Stop;

    ELSE

      --Error msg
      Put_Line
      ("Run_Time_Analysis: Analyze_Operator_Execution_Data: STOP < START
                                                                    for");
      Put(Operator_Data(I).Operator_Name);

    END IF;

    -- Send results to be analyzed
    Analyze_Results_Data(Result_Data, Operator_Data(I));

  END LOOP;

END Analyze_Operator_Execution_Data;


--*****************************************************************--
-- Get_Operator_Timing_Data                                        --
--                                                                 --
-- This procedure is called by the scheduler program to get the planned --
-- run time data from CAPS SCHEDULER module. This scheduled run time --
-- data includes predetermined execution starting and stopping time of the --
-- CAPS operator.                                                  --
*****************************************************************--
PROCEDURE Get_Operator_Timing_Data
            (Result_Data : IN OUT Run_Time_Record;
             Operator_Data : IN OUT Run_Time_Record) IS

BEGIN--Get_Operator_Timing_Data

  IF (Operator_Data.Planned_Stop > Operator_Data.Planned_Start) THEN

    -- Calculate Planned Run Time
    Result_Data.Planned_Run_Time :=
        (Operator_Data.Planned_Stop - Operator_Data.Planned_Start);

    Operator_Data.Planned_Run_Time := Result_Data.Planned_Run_Time;

  ELSE
```

```
    --Error msg
    Put(Operator_Data.Operator_Name);
    Put_Line("Run_Time_Analysis: Get_Operator_Timing_Data: STOP < START");
   END IF;

  END Get_Operator_Timing_Data;

--*************************************************************--
-- Analyze_Results_Data                                        --
--                                                             --
-- This procedure is called within the Run_Time_Analysis module to --
-- perform an analysis upon the recently retrived CAPS operator --
-- run time data. This run time data consists of the planned execution --
-- starting and stopping, as well as actual execution start and stop --
-- times of this CAPS operator.                                --
--*************************************************************--
  PROCEDURE Analyze_Results_Data(Result_Data : IN OUT Run_Time_Record;
                                 Operator_Data : IN OUT Run_Time_Record) IS

  BEGIN--Analyze_Results_Data

    --Get total run time
    Result_Data.Total_Run_Time :=
      Operator_Data.Actual_Run_Time + Result_Data.Total_Run_Time;

    --Determine average run time
    Result_Data.Average_Run_Time :=
        (Result_Data.Total_Run_Time / Result_Data.Execution_Count);


    --Check for new minimum run time
    IF (Result_Data.Actual_Run_Time <
      (To_Time_Span(Duration(Result_Data.Minimum_Run_Time)))) THEN

      Result_Data.Minimum_Run_Time :=
              (Float(To_Duration(Result_Data.Actual_Run_Time)));

    END IF;

    --Check for new maximum run time
    IF (Result_Data.Actual_Run_Time > Result_Data.Maximum_Run_Time) THEN

      Result_Data.Maximum_Run_Time := Result_Data.Actual_Run_Time;

    END IF;
```

```
    --Find planned/actual run time difference
    IF (Operator_Data.Actual_Run_Time >
                        Operator_Data.Planned_Run_Time) THEN

      Result_Data.Run_Time_Difference :=
         Operator_Data.Actual_Run_Time - Operator_Data.Planned_Run_Time;

      Result_Data.Total_Timing_Error :=
                  Result_Data.Total_Timing_Error + 1;

      IF (Operator_Data.Operator_Type = "Soft") THEN
        Put("Timing error from operator ");
        Put(Operator_Data.Operator_Name);
        New_Line;
      END IF;

    ELSE

      Result_Data.Run_Time_Difference :=
        Operator_Data.Planned_Run_Time - Operator_Data.Actual_Run_Time;

    END IF;

    --Ship execution data
    Send_Operator_Run_Time_Calculation_Data(Result_Data);

  END Analyze_Results_Data;


--*****************************************************************--
-- Check_Soft_Task_Error                                           --
--                                                                 --
-- This procedure is called within the New_Thermostat_SRTScheduler --
-- module to determine if the soft task finishes execution within its --
-- deadline and if it does not, an error message is displayed by the --
-- procedure.                                                      --
--*****************************************************************--


  PROCEDURE Check_Soft_Task_Error(Result_Data : IN OUT Run_Time_Record;
                      Operator_Data : IN OUT Run_Time_Record) IS

  BEGIN--Check_Soft_Task_Error

    Operator_Data.Completion_Time := Operator_Data.Arrival_Time +
                        Operator_Data.Response_Time;
```

```
IF (Operator_Data.Completion_Time > Operator_Data.Deadline) THEN

  IF (Result_Data.Miss_Flag = False) THEN
    Result_Data.Miss_Flag := True;

    Result_Data.Consecutive_Misses := Result_Data.Consecutive_Misses + 1;

    IF (Result_Data.Consecutive_Misses >
       Result_Data.Max_Consecutive_Misses) THEN
       Result_Data.Max_Consecutive_Misses :=
              Result_Data.Consecutive_Misses;
    END IF;

  ELSE
    Result_Data.Consecutive_Misses := Result_Data.Consecutive_Misses + 1;

    IF (Result_Data.Consecutive_Misses >
       Result_Data.Max_Consecutive_Misses) THEN
       Result_Data.Max_Consecutive_Misses := Result_Data.Consecutive_Misses;
    END IF;

  END IF;

  Result_Data.Total_Missed_Deadlines := Result_Data.Total_Missed_Deadlines + 1;

  Result_Data.Individual_Tardiness := Operator_Data.Completion_Time -
                                      Operator_Data.Deadline;

  Result_Data.Total_Tardiness := Result_Data.Total_Tardiness +
                    Result_Data.Individual_Tardiness;

  IF (Result_Data.Individual_Tardiness > Result_Data.Maximum_Tardiness) THEN
    Result_Data.Maximum_Tardiness := Result_Data.Individual_Tardiness;
  END IF;

  Result_Data.Average_Tardiness := Result_Data.Total_Tardiness /
                  Result_Data.Total_Missed_Deadlines;

  Put(Operator_Data.Operator_Name);
  Put(" missed its deadline.");
  New_Line;

ELSE

  Result_Data.Individual_Tardiness := Time_Span_Zero;
```

```
      Result_Data.Miss_Flag := False;

      Result_Data.Consecutive_Misses := 0;

    END IF;

  END Check_Soft_Task_Error;

--*************************************************************--
-- Send_Operator_Run_Time_Calculation_Data                     --
--                                                             --
-- This procedure is called by the scheduler program to transmit run time    --
-- calculations to Run_Time_Results module. This scheduled run time data     --
-- includes predetermined execution starting and stopping time of the        --
-- CAPS operator as well as actual execution start and stop times of this    --
-- operator.                                                   --
--*************************************************************--
  PROCEDURE Send_Operator_Run_Time_Calculation_Data
                    (Result_Data:IN OUT Run_Time_Record) IS

  BEGIN--Send_Operator_Run_Time_Calculation_Data

    --Send the execution data to Run_Time_Results module
    Get_Operator_Run_Time_Calculation_Data(Result_Data);

  END Send_Operator_Run_Time_Calculation_Data;

END Run_Time_Analysis;
```

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.....2
   8725 John J. Kingman Road, Ste 0944
   Ft. Belvoir, VA 22060-6218

2. Deniz Kuvvetleri Komutanligi 2
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

3. Deniz Harp Okulu Komutanligi 1
   Kutuphane
   Tuzla
   Istanbul, TURKEY

4. Dudley Knox Library.....2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

5. Chairman, Department of Computer Science 1
   Code CS
   Naval Postgraduate School
   Monterey, CA 93943

6. Dr. Mantak Shing, Code CS/Sh 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

7. Dr. Valdis Berzins, Code CS/Be 1
   Computer Science Department
   Naval Postgraduate Department
   Monterey, CA 93943

8. CDR Michael J. Holden, USN Code UW/Hm 1
   Underwater Warfare Department
   Naval Postgraduate School
   Monterey, CA 93943

9. LTJG Omer Korkut 4
Hosdere Caddesi
Resat Nuri Sokak 69/29
Yukari Ayranci
Ankara, TURKEY